

Evolution in Model-Driven Software Product-line Architectures

Gan Deng¹, Douglas C. Schmidt¹, Aniruddha Gokhale¹, Jeff Gray², Yuehua Lin², and Gunther Lenz³

¹Department of EECS
Vanderbilt University
Nashville, TN 37203, USA
{gan.deng, d.schmidt, a.gokhale} @vanderbilt.edu

²Computer and Information Sciences Department
University of Alabama at Birmingham
Birmingham, AL 35294, USA
{gray, liny} @ cis.uab.edu

³Software Engineering Department
Siemens Corporate Research
Princeton, NJ 08540, USA
lenz.gunther@siemens.com

1. Introduction

Software *product-line architectures* (PLAs) are a promising technology for industrializing software-intensive systems by focusing on the automated assembly and customization of domain-specific components, rather than (re)programming systems manually (Clements et. al, 2001). A PLA is a family of software-intensive product variants developed for a specific domain that share a set of common features. Conventional PLAs consist of *component frameworks* (Szyperski., 2002) as core assets, whose design captures recurring structures, connectors, and control flow in an application domain, along with the points of variation explicitly allowed among these entities. PLAs are typically designed using *scope/commonality/variability* (SCV) analysis (Coplien et. al, 1998), which captures key characteristics of software product-lines, including: (1) *scope*, which defines the domains and context of the PLA, (2) *commonalities*, which name the attributes that recur across all members of the product family, and (3) *variabilities*, which contain the attributes unique to the different members of the product family.

Motivating the need for model-driven software product-line architectures. Despite improvements in third-generation programming languages (such as C++, Java and C#) and runtime platforms (such as CORBA, J2EE and Web Services middleware), the levels of abstraction at which PLAs are developed today remains low-level relative to the concepts and concerns within the application domains themselves, such as manually tracking the library dependency or ensuring component composition syntactical and semantic correctness. A promising means to address this problem involves developing PLAs using *model-driven engineering* (MDE) (Schmidt, 2006), which involves systematic use of models as key design and implementation artifacts throughout the software lifecycle. MDE represents a design approach that enables description of the essential characteristics of a problem in a manner that is decoupled from the details of a specific solution space (e.g., dependence on specific OS, middleware or programming language).

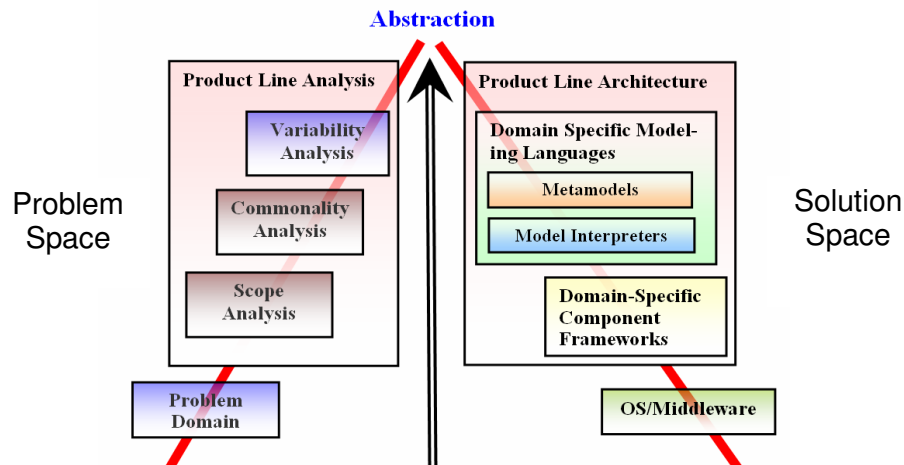


Figure 1: Using DSMLs and Domain-specific Component Frameworks to Enhance Abstraction and Narrow the Gap between Problem and Solution Space of Software-Intensive Systems

As shown in Figure 1, MDE-based PLAs help raise the level of abstraction and narrow the gap between the problem space and the solution space of software-intensive systems by applying the following techniques:

- *Domain-specific Modeling Languages (DSMLs)*. A DSML (Gray et al., 2007) consists of *metamodels* and *model interpreters*. A metamodel is similar to the grammar corresponding to a programming language that defines a semantic type system that precisely reflects the subject of modeling and exposes important constraints associated with specific application domains. Model interpreters can read and traverse the models, analyze them, and help create the executable system based on these models. DSMLs help automate repetitive tasks (Gray et al., 2006) that must be accomplished for each product instance, including generating code to glue components or synthesizing deployment and configuration artifacts for middleware platforms and the underlying operating systems (Balasubramanian et al., 2006).
- *Domain-specific Component Frameworks*. Through SCV analysis, object-oriented extensibility capabilities are often used to create domain-specific component frameworks¹, which factor out common usage patterns in a domain into reusable platforms (Clements et al., 2001). These platforms, in turn, help reduce the complexity of designing DSMLs by simplifying the code generated by their associated model interpreters and addressing the product-line specific functional and systemic concerns, including quality of service (QoS) concerns, such as latencies, throughput, reliability, security, and transactional guarantees.

MDE helps software developers explore various design alternatives that represent possible configurations for a specific instance of the product family. For example, a product instance ultimately needs to be deployed into a specific target running environment, where all software components must be deployed and mapped to available hardware devices and configured properly based on the specific software/hardware capabilities of the devices. If the PLAs are intended for use with different hardware devices, however, the mappings between the software components and hardware devices cannot not be known *a priori* when the software PLAs are developed. Instead of analyzing every product instance individually and manually writing source code or scripts repetitively for every different target execution environment, an MDE-based approach to PLA deployment and configuration automates such repetitive and labor-intensive tasks by integrating domain knowledge and expertise into metamodels and model interpreters. Hence, a DSML infuses intelligence into domain models, which helps address many “what if” problems, such as “what glue code or configuration scripts must be written if the product is to be deployed into an environment with XYZ requirements?” These “what if” scenarios help developers understand the ramifications of design choices of software-intensive systems at a higher level of abstraction than changing source code manually at the implementation level.

Challenges with evolution of model-driven software product-line architectures. Although an MDE-based approach helps improve productivity of software-intensive systems by raising the level of abstraction through composition of DSMLs and domain-specific component frameworks, it is hard to evolve software PLAs by incorporating new requirements. Examples of such requirements include using new software platforms or applying the current PLA in a new use case that may impose a different set of concerns than those handled by the current PLA. Consequently, in addition to assisting in the exploration of design alternatives among product instances, an MDE-based PLA technology must also address the *domain evolution problem* (Macala et al., 1996), which arises when existing PLAs must be extended and/or refactored to handle unanticipated requirements.

Depending on the scopes of the DSMLs and domain-specific component frameworks, unanticipated requirements can be either functional requirements or non-functional requirements, or both. For example, consider an MDE-based PLA that is available on two different component middleware technologies, such as Enterprise Java Beans (EJB) (Sun Microsystems, 2001) and CORBA Component Model (CCM) (OMG, 2006). A goal of a DSML that supports PLA is to selectively use the technologies within a product instance based on the system requirements. The metamodel of the DSML must define proper syntax and semantics to represent both component middleware technologies. With domain evolution, if the CCM technology must be replaced by another emerging middleware technology such as Web Services, the MDE-based PLA must evolve accordingly to satisfy the new requirements, i.e., new syntax and semantics must be introduced into the metamodel and new domain-specific component frameworks must be developed based on the emerging Web Services technology.

Unfortunately, adding new requirements to MDE-based PLAs often causes invasive modifications to the PLAs in the DSMLs and component frameworks if DSMLs and component frameworks were not initially designed to be extensible to incorporate such new requirements. Conventional MDE tools do not handle the domain evolution problem effectively because they require significant handcrafted changes to existing PLAs, at both the component framework level and the DSML level. The domain evolution problem is particularly hard because the coupling of architecture and infrastructure concerns often *crosscut* the component framework layer and the DSML layer (Deng et al., 2006) within a PLA.

¹. Unless stated otherwise, we use the two terms “domain specific component frameworks” and “component frameworks” inter-changeably throughout this chapter.

Moreover, changes made on metamodels in a PLA often invalidate existing domain models based on previous versions of the metamodels (Sprinkle et al., 2004), which makes the evolution process of model-driven software PLAs hard. Other examples of this problem occur in programming language or object-oriented framework design, where changes to a grammar or class hierarchy for a programming language or framework may introduce errors in existing legacy source code (Klusener et al. 2005). Another example is schema evolution in a database, where changes to a database schema may render the contents of the database useless (Roddick 1992). Just like legacy source code and contents of database, domain model are crucial assets of an organization, so they must be well handled during the metamodel evolution process.

From these observations, there are many complexities involved when MDE-based software PLAs need to evolve. Although software developers can manually update their metamodels, domain models, and component frameworks for small-scale systems, this approach is clearly tedious, time consuming, error-prone, and non-scalable for software-intensive systems.

Solution → Systematic PLA evolution with automated domain model transformation. To address these challenges, a layered and compositional architecture is needed to modularize system concerns and reduce the effort associated with domain evolution. With the help of this architecture, different layers of PLAs can evolve systematically and tool supported domain model evolution also becomes feasible. The overall approach can be characterized in the following ordered steps:

1. The first step deals with component framework evolution. Since component frameworks provide core functionalities to the product instances, they have the most direct impact on the PLAs. As a result, whenever PLAs need to incorporate new requirements, component frameworks must first be refactored. To reduce the impact of such evolution outside the component frameworks, the key point is using pattern-oriented software architecture (Gamma 1995, Schmidt 2000).
2. The second step deals with metamodel evolution. Since metamodels are used to define type systems of particular domains based on proper language syntax, a language can be decomposed into smaller units to localize the evolution impact, and allow such smaller units to be composed to form the new metamodel.
3. The third step deals with the domain model transformation. This step applies automated model transformation techniques to specify model-to-model transformation rules that define metamodel changes. The application of automated model transformation alleviates many tedious, time consuming, and error-prone tasks of model-to-model transformation to reduce the complexity of PLA evolution. In particular, when an existing DSML in a PLA is changed, the domain models defined by this DSML can be migrated automatically to the new DSML by apply a set of model transformation rules.

While the three-step approach above could be applied to any model-driven software PLAs, this chapter focuses on distributed real-time embedded (DRE) PLAs, which are among the most difficult software-intensive systems to develop since such systems have limited resources and must communicate via the network to meet stringent real-time quality-of-service (QoS) assurance and other performance requirements.. A representative software-intensive DRE system is used throughout the chapter as a case study to describe how to evolve PLAs systematically and minimize human intervention. Along with presenting the approach for domain evolution of MDE-based PLAs, the chapter also describes key concepts, such as model-driven engineering, product-line architectures, and model transformations that are important for developing and evolving PLAs for large-scale software-intensive systems.

The remainder of this chapter is organized as follows: Section 2 evaluates related work that supports evolution of software PLAs for DRE systems and compares it with our approach; Section 3 describes a conceptual architecture of MDE-based PLAs for DRE systems and defines the key elements in this architecture as background of this chapter; Section 4 introduces a representative case study of a PLA for avionics mission computing used throughout the chapter; Section 5 describes the challenges we faced evolving model-driven PLAs and the solutions to address these challenges; Section 6 outlines future trends of MDE-based PLAs for DRE systems; and Section 7 presents concluding remarks and lessons learned.

2. Related Work

This section surveys the technologies that provide solutions to MDE-based software PLA evolution for software-intensive systems. The related work has been categorized along two dimensions based on the syntax of the modeling mechanism the software PLA evolution relies on, i.e., a graphical based modeling approach or a text-based modeling approach.

2.1 Graphical Modeling Approaches

A UML metamodel for software PLA evolution (Mens et al., 2000) has been developed based on the concept of *evolution contract*. The idea of an evolution contract is that when incremental modifications and evolution of

software artifacts are made on a software product line, a formal contract must be defined between the provider and the modifier. The purpose of the contract is to define the evolution behavior formally. A UML metamodel has been defined to capture the formal evolution contract. This offers a generic MDE-based mechanism for dealing with unanticipated evolution. By documenting model evolution through formal models, incompatibilities or undesired behavior across different modeling artifacts can be detected when models are upgraded, or when different software developers independently make changes to the same or related parts of a model. This approach allows conflicts to be detected regardless of the specific kind of model that is under consideration. The approach has been integrated into third-party CASE tools, such as IBM Rational Rose (IBM, 2007).

KobrA (Atkinson, 2002) is another approach based on UML for component-based software PLAs that support model-driven representation of components. In this method, evolution management in software PLAs is divided into three activities, i.e., configuration management, change management, and maintenance planning. Configuration management in KobrA is a static method for bringing together different artifacts within a PLA. Change management consists of techniques to evaluate evolution requests. The use of appropriate formal change operators to evolution requests assists in traceability within change propagations in a PLA. Maintenance planning is responsible for constructing infrastructure for the change and configuration management activities. The idea of KobrA is based on a change-oriented model, i.e. new versions are obtained from changes applied to some artifacts in the product line. To support the evolution in KobrA, the *evolution graph* technique (Atkinson, 2002) is proposed to capture version histories of different artifacts of the PLA and trace the dependencies.

Another technique similar to the evolution graph is called *design decision tree* (DDT) (Ran et al., 1996), which is a formal approach to incrementally document, refine, organize and reuse the architectural knowledge for software design. The formalism is a hierarchical organization of design patterns that is a partial ordering of design decisions put in the context of the problem requirements and the constraints imposed by earlier decisions. This model integrates architectural knowledge of software design into a software development process. A DDT contains system-wide design information in a form that can be used to analyze change requests and determine their impact on system structure. Because the tree is maintained throughout the lifecycle of a PLA, it can be used as the main repository of design knowledge (Karhinen, 1998). Such a repository can be used to analyze the impact of new requirements to the existing requirement space and to investigate the changes that different implementation strategies may cause to the system structure, which makes it possible to classify different options and to react to them and analyze their architectural implications.

Summary. The related work described above adopts a domain-independent modeling technique to capture the software PLA evolution requirements either explicitly or implicitly. Our approach is similar to these related works in the sense that they all provide visualization capabilities through graphical modeling tools for PLAs. Our approach, however, uses a domain-specific modeling technique that adds additional abstractions representing domain concepts to the modeling languages that are not available in general-purpose domain-independent modeling languages such as UML. DSMLs thus require less effort and fewer low-level details to specify a given system (Tolvanen et al., 2005).

2.2 Text-based Modeling Approaches

Architectural Description Language (ADL) is an important technique in this dimension that facilitates software PLA evolution. The Mae environment (Hoek et al., 2001), for example, uses ADL to facilitate incremental evolution by capturing all changes made to any architectural elements within a PLA. A key concept in the Mae environment is a system model that allows architectural concepts and configuration management to be mapped with each other through ADL syntax, i.e., the ADL allows users to describe what and how the changes should be made to the model. The essence of the approach lies in the use of this model to integrate change management concepts (such as revisions, variants, and configurations) with architectural concepts (such as components, connectors, subtypes, and styles) through ADL descriptions. By mapping the generic system model onto a specific ADL, the design analyses of a software PLA and its evolution can be adapted for the purpose of maintaining the consistency of the architectural configurations captured by the model.

Similar to the idea of the Mae environment, the Koala Component Model (Ommering et al., 2000) also uses ADL to explicitly describe the architecture and provides a platform-centric approach to the design of PLAs for consumer electronics software. Specifically, the Koala Component Model allows variability options to be modeled explicitly via a property mechanism. Using a third-party versioning system, Koala can be used to capture the evolution of a PLA.

XADL (Dashofy et al., 2003) is an XML-based ADL that is constructed from a set of extensible XML schemas. XADL also defines a set of associated libraries that provide a programmatic interface to XADL documents, and provide runtime facilities to create, store, and modify XADL documents. XADL and its associated libraries pro-

vide three important benefits for the purposes of supporting software PLA evolution: (1) the core of the XADL language supports variability in both space and time; in XADL, variabilities of artifacts are a natural and integral part of the language, (2) the language can be extended, which allows individual activities in the lifecycle to be able to attach additional information, and (3) the library provides a generic interface to easily access XADL documents, which supports the rapid construction of new tools supporting PLA evolution.

Summary. The related work described in this section all use text-based languages (such as structural languages or XML) to either explicitly capture the PLA evolution activities, or implicitly associate the evolution requirements with actual software components. Our approach is similar to this dimension of the related work in the sense that PLA evolution can be captured through the software PLA architecture itself, rather than through a separate dedicated language.

2.3. Hybrid Approaches

Some technologies span both text-based and graphical-based approaches. A well-know example in this category is called QVT (Query/View/Transformation) (OMG 2005b), which is the OMG standard for model-to-model transformations. This technology provides a standard language to transform UML or custom model types from one type to another. It accepts XMI as input and output. Typical usage scenarios include automating transformation of a high-level design model into a more detailed model, transforming a UML model into a custom data model, or transforming one custom model type into another. The core benefits of this feature set are a standards-based language to express common model transformations with traceability, which provides repeatable results.

Summary. The specification of QVT defines both graphical syntax and textual syntax for the transformation language, but so far there still lacks a full implementation of the specification. Moreover, while QVT is restricted to only XMI to XMI transformations, our approach does not have this restriction so it can exploit any internal representation of the DSMLs.

3. An MDE-based Product Line Architecture for DRE Systems

This section introduces an architecture of a MDE-based product line architecture for software-intensive DRE systems, focusing on the design concepts, common patterns, and software methodology. An MDE-based design and composition approach for DRE systems entails the combination of DSMLs with reusable component frameworks. Figure 2 illustrates the high-level design principles and an overall architecture of an MDE-based PLA solution for software-intensive DRE systems that exploits a *layered* and *compositional* approach. This architecture takes advantage of layering and composition design principles (Krueger et al., 2006) to make the associated PLAs easier to develop and evolve than *ad hoc* approaches.

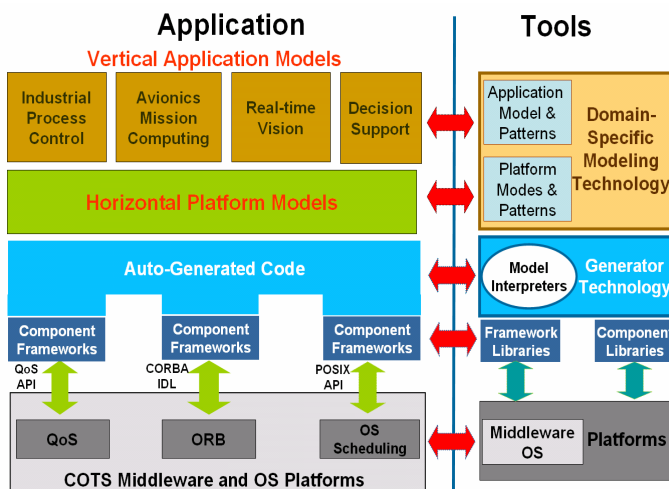


Figure 2. MDE-Based Product-line Architecture for DRE Systems

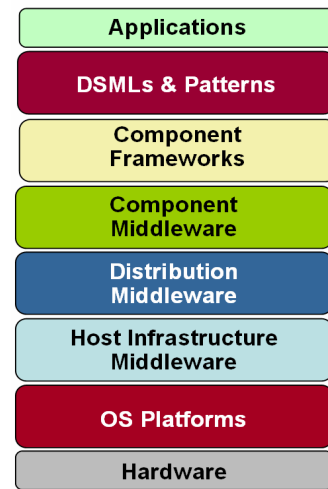


Figure 3. OS, Middleware, DSML and Application Layer Relationships

As shown in Figure 2, the PLA architecture is based on a core set of COTS middleware and OS platforms, component frameworks and domain specific modeling languages. The right side of the figure shows the technologies available to implement the design artifacts on the left side. For example, the “Generator Technology” shown on the right can be used to build model interpreters that automatically generate code to bridge the gap between models and component frameworks.

The remainder of this section introduces and defines key terms and concepts in the architecture shown in Figure 2.

Commercial-off-the-shelf (COTS) middleware and OS platforms provide the infrastructure upon which DRE systems run. Many DRE systems are based on OS platforms with real-time scheduling capabilities. Examples of such OS platforms include *VxWorks* (Wind River Systems, 1998), *Timesys Linux* (Timesys, 2002), and *Windows CE* (Microsoft, 2007). Middleware is an enabling technology that allows multiple processes running on one or more machines to interact across a network. Middleware can be further decomposed into multiple layers (Schmidt et al., 2002), such as those shown in Figure 3 and described below:

- **Host Infrastructure Middleware.** The host infrastructure layer resides directly atop the operating system and provides a set of higher-level APIs that hide the heterogeneity of different operating systems and network protocols. The host infrastructure layer provides generic services to the upper middleware layers by encapsulating functionality that would otherwise require much tedious, error-prone, and non-portable code, such as socket programming and thread manipulation primitives. Examples of such middleware include ACE (Schmidt, 1993), Real-time Java (Bollella et al., 2000) and Rocks (Zandy, 2002).
- **Distribution Middleware.** The distribution layer resides atop the host-infrastructure layer and provides high-level programming abstractions, such as remote object operations. Using the distribution layer, a developer can write a distributed application in a similar way to a stand-alone application. CORBA 2.x (OMG, 2003), DCOM (Microsoft, 2000), Java RMI (Sun Microsystems, 2000) and Data Distribution Service (DDS) (OMG, 2004) are the main solutions to distribution middleware.
- **Component Middleware.** The component middleware layer resides atop the distribution middleware layer and adopts the component-based software engineering approach to allow maximum reuse of software components. Component middleware also provides mechanisms to configure and control key distributed computing aspects, such as connecting event producers to event consumers and managing transactional behavior, separate from the functional aspects of the application. Examples of component middleware platforms include Enterprise Java Beans (EJB) (Sun Microsystem, 2001) and OMG Corba Component Model (CCM) (OMG, 2005).

Because many DRE systems require a loosely-coupled distribution architecture to simplify extensibility, COTS middleware typically provides event-based publish/subscribe communication mechanisms, which help reduce ownership costs by defining clear boundaries between the components in the application. Such mechanisms reduce dependencies and maintenance costs associated with replacement, integration, and revalidation of components. COTS middleware and OS platforms are designed to maintain the commonality, portability, reusability, and applicability of software for different domains.

Component frameworks provide reusable domain-specific building blocks for PLAs of DRE systems. As illustrated in Figure 3, component frameworks reside atop COTS middleware and OS platforms. The key difference between component frameworks and component middleware is that the latter is domain independent while the former is domain-specific. Component frameworks define “semi-complete” applications that embody domain-specific object structures and functionality to raise the level of abstraction at which the software product instance is composed, and offer product-line specific environments to capture the variabilities. Components in such a framework coordinate with each other to provide core functionalities for a family of related applications. Complete applications can be composed by inheriting from and/or instantiating framework components. .

Examples of component frameworks include the Boeing Bold Stroke product line architecture (Schulte, 2003) in the avionics mission computing domain and Siemens Building Technology APOGEE product line architecture (SIEMENS 2007) in the building automation domain. For example, the Boeing Bold Stroke PLA supports many Boeing product variants using a component-based platform. The Boeing Bold Stoke PLA supports systematic reuse of mission computing functionality and is configurable for product-specific functionality and execution. The philosophy of component frameworks is to develop reusable components that are well-defined and have specific use contexts and variability points, which helps reduce the effort associated with using low-level middleware interfaces or OS APIs.

Domain-specific modeling languages (DSMLs) and patterns facilitate the model-based design, development, and analysis of DRE systems. Figure 4 shows how DSMLs and patterns can be combined with component frameworks to build product instances. A DSML can represent either a vertical application domain model (specific to concerns within a specific industry or domain) or a horizontal model (generic to concerns that span several domains).

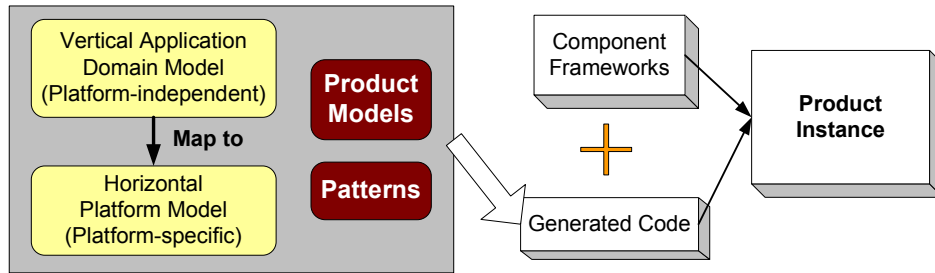


Figure 4. Integration of Domain-specific Modeling and Component Frameworks

Vertical application domain models address the problems arose within a particular domain, and they are often modeled in a platform-independent manner (Frankel, 2003). Examples of such vertical application domains include industrial process control, telecommunications, and avionics mission-critical systems. Some DSML examples developed for vertical domains include the *Saturn Site Production Flow* (SSPF), which is a manufacturing execution system serving as an integral and enabling component of the business process for an automotive factory (Long et al., 1998). Another example is the *Embedded System Modeling Language* (ESML) (Karsai et al., 2002), which models mission computing embedded avionics applications in the Boeing Bold Stroke PLA.

Horizontal platform domain models are also called platform-specific models (Frankel, 2003). A platform-specific model is a model of a system that is linked to a specific technological platform (e.g., a specific middleware platform, operating system or database). An example of a DSML for horizontal platforms is the *Rhapsody* modeling environment (iLogix, 2006), which allows application generation for embedded software platforms based on many real-time operating systems. Other examples of DSMLs for horizontal platforms include the *Platform Independent Component Modeling Language* (PICML) (Balasubramanian et al., 2005a) and J2EEML (White et al., 2005), which facilitate the development, deployment, and configuration of QoS-enabled component-based DRE systems based on CCM and EJB, respectively.

The main idea is that it should be possible to use a model transformation technique to transform vertical application domain models to a horizontal platform domain model. Regardless of whether the DSMLs target horizontal or vertical domains, *model interpreters* can be used to generate various artifacts (such as code and metadata descriptors for deployment and configuration), which can be integrated with component frameworks to form executable applications and/or simulations. Key advantages of using DSMLs and patterns in PLAs are to rigorously capture the key roles and responsibilities of a product instance and help automate repetitive tasks that must be accomplished for each product instance.

As summary, an MDE-based PLA for software-intensive systems must based on an architecture that adheres to well documented principals of architectural design with a clear separation of commonalities and appropriate provisions for incorporating variations by integrating vertical/horizontal DSMLs, component frameworks, middleware and OS platforms. In this architecture, MDE technologies are used to model PLA features and glue components together; for example, they could be utilized to synthesize deployment artifacts for standard middleware platforms (Balasubramanian et al., 2006).

4. Overview of the Boeing Bold Stroke PLA and EQAL MDE Tool

This section introduces a case study based on a real-time avionics mission computing product line called Boeing Bold Stroke and describes the structure and functionality of the *Event QoS Aspect Language* (EQAL) MDE tool based on this product line. The Boeing Bold Stroke PLA supports many Boeing product variants (e.g., F/A-18E, F/A-18F, F-15E, and F-15K) using a component-based publish/subscribe pattern (Gamma 1995).. The EQAL MDE tool is intended to reduce many complexities associated with the integration, deployment and configuration of different implementations of publish/subscribe mechanism. The Bold Stroke PLA and its associated models in EQAL will serve as the case study throughout this chapter.

4.1 Overview of Boeing Bold Stroke Product Line Architecture

Figure 5 illustrates the Boeing Bold Stroke PLA (Sharp, 1999), which was developed by Boeing in the mid-1990s to support systematic reuse of avionics mission computing functionality and is configurable for product-specific functionality (such as heads-up display, navigation, and sensor management) and execution environments (such as different networks/buses, hardware, operating systems, and programming languages) for a variety of military aircraft. Bold Stroke is a very complex framework with several thousand components implemented in several million lines of C++ code.

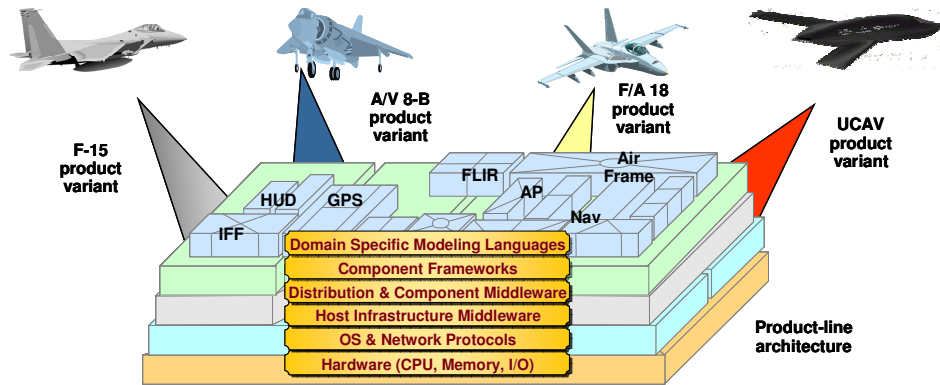


Figure 5. Boeing Bold Stroke Product Line Architecture

The Boeing Bold Stroke architecture contains set of event-driven component-based component frameworks built atop (1) The ACE ORB (TAO) (Schmidt et al., 1998), which implements key Real-time CORBA (OMG, 2005) features, and (2) TAO's Real-time Event Service (Harrison et al., 1997), which implements the publish/subscribe architectural pattern. Bold Stroke uses a Boeing-specific component model called PRISM (Roll, 2003), which implements a variant of the CORBA Component Model (CCM) atop TAO.

Following the CCM specification, PRISM defines the following types of ports, which are named interfaces, and connection points components used to collaborate with each other:

- **Facets**, which define named interfaces that process method invocations from other components.
- **Receptacles**, which provide named connection points to facets provided by other components.
- **Event sources and event sinks**, which indicate a willingness to exchange event messages with one or more components via event channels.

Bold Stroke is a representative PLA for DRE systems in the real-time avionics mission computing domain. Its event based communication architecture employs a control flow/data flow (Sharp, 1999) principle, where control flow represents the movement of execution through a software system, while the data flow represents the movement of data through a software system. Depending on requirements, different product variants in the Boeing Bold Stroke PLA may require different levels of QoS assurance for event communication, including timing constraints, event delivery latency, jitter, and scalability. Even within the same product variant, different levels of QoS assurance must be ensured for different communication paths, depending on criticality of the data. For example, the communication path between a collision radar component and the LED display component must have much more stringent timeliness deadline requirements than regular GPS components and navigation display components.

To alleviate the complexity in provisioning the event-based publish/subscribe services and their QoS assurance in the Boeing Bold Stroke PLA, we designed an MDE-based tool called the Event QoS Aspect Language (EQAL) that can automate and simplify the integration of publish/subscribe services into QoS-enabled component-based systems.

4.2 Overview of the EQAL MDE Tool

One core part of the EQAL MDE tool is the EQAL DSML (Edwards et al., 2004), which is implemented using the Generic Modeling Environment (GME) (Lédeczi, 2001). The GME is a toolkit that supports the development of DSMLs, as described in Sidebar 1. The EQAL DSML provides an integrated set of metamodels, model interpreters, and standards-based component middleware that allow DRE system developers to visually configure and deploy event-based communication mechanisms in DRE systems via models instead of programming them manually. The EQAL DSML is an example that supports a horizontal platform domain; i.e., it is not restricted to a particular vertical application domain, but instead can be leveraged by multiple vertical domains. In this case study, we describe how EQAL was applied to the Bold Stroke avionics mission computing PLA.

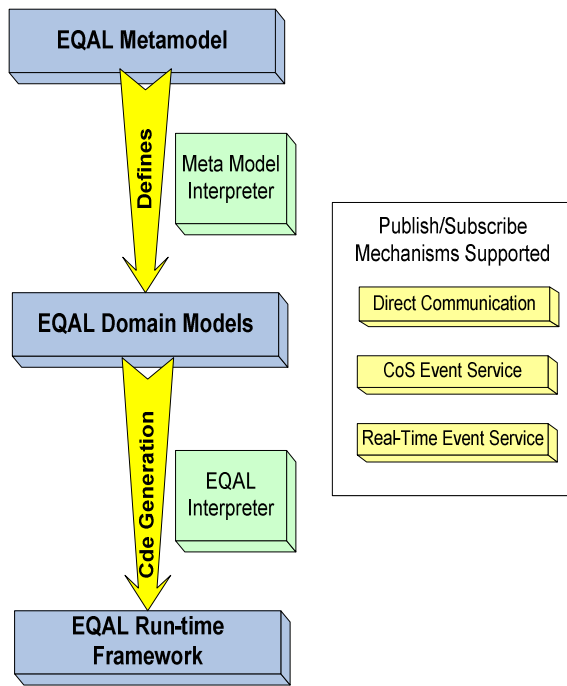


Figure 6. EQAL MDE Tool Architecture

Sidebar 1: Generic Modeling Environment (GME)

The Generic Modeling Environment (GME) is a meta-modeling tool for creating and evolving domain-specific models. GME allows developers to create domain-specific modeling languages (DSMLs) that capture multiple aspects of a domain, such as structural, functional and behavior aspects, and their semantics (i.e., constraints) to ensure correctness of constructed models. End-users of the DSML then in turn use GME to construct models realized by the DSML.

One of the main features of GME is the ability to implement components that operate on target models of a DSML. GME supports the following components:

- **Interpreters**, which are invokeable components specific to a DSML, such as parsing a DSML to generate a configuration file.
- **Add-ons**, which are event-driven components specific to a DSML, such as responding to the creation of new elements to initialize attributes.
- **Plug-ins**, which are invokeable components applicable to any DSML, such as providing a tabular view of the model in any DSML.

The main goal of the components is to provide both domain-specific, and non domain-specific, functionality that cannot be captured in a metamodel. GME is available as open-source for download (GME, 2007).

As shown in Figure 6, EQAL is a layered architecture that supports several types of abstractions, which are subject to change stemming from domain evolution as explained below:

- The *bottom layer* in the architecture is the EQAL Runtime Framework, which is a portable, OS-independent middleware framework based on light-weight CCM (OMG, 2004b). The EQAL Runtime Framework provides an extensible way to deploy various event-based publish/subscribe mechanisms, including a two-way event communication mechanism based on direct method invocation instead of using a mediator channel.
- The *middle layer* in the EQAL architecture is a set of domain models that represent instances of the modeled DRE systems. These models are created using the EQAL DSML and are used to capture the structural and behavioral semantic aspects of event-based DRE systems.
- The *top layer* of the EQAL architecture consists of a metamodel that enables developers to model concepts of event-based DRE systems, including the configuration and deployment of various publish/subscribe services. This layer also contains several model interpreters that synthesize different types of configuration files that specify QoS configurations, parameters, and constraints, such as the threading model for event dispatching, event filtering configuration, and event channel federation configurations (Edwards et al., 2004). The EQAL interpreters automatically generate publish/subscribe service configuration files and service property description files needed by the underlying EQAL Runtime Framework and selected middleware.

As shown in Figure 7, EQAL allows DRE system deployers to create and synthesize publish/subscribe QoS configurations and deployments via graphical models (i.e., EQAL domain models) that are much easier to understand and analyze than hand-crafted code. During the modeling phase, EQAL ensures that dependencies between configuration parameters are enforced by declaring constraints on the contexts in which individual options are valid (e.g., priority-based thread allocation policies are only valid with component event connections that have assigned priorities). EQAL can then automatically validate configurations and notify users of incompatible QoS properties during model validation, rather than at component deployment and run-time. The generated XML-based QoS configuration and deployment descriptors can then be fed into deployment and configuration runtime tools to deploy and configure the components and real-time event channels within the Boeing Bold Stroke.

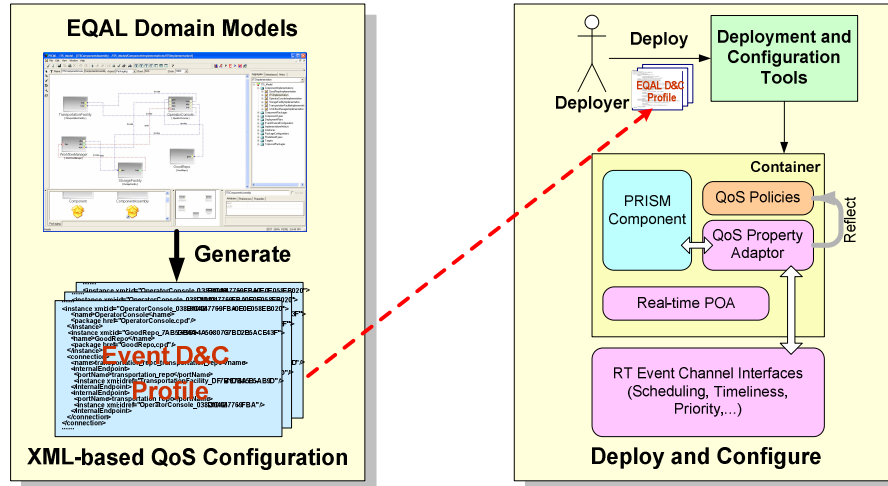


Figure 7. Code Generation from EQAL Domain Model

5. Support MDE-based PLA Evolution when Facing Domain Evolution

This section examines the following challenges associated with evolving MDE-based PLAs:

1. Challenges stemming from capturing new requirements into existing MDE-based PLAs for DRE systems.
2. Challenges stemming from migrating existing domain models with MDE-based PLA evolution.

For each challenge, we explain the context in which the challenge arises and identify key problems that must be addressed. Many of these challenges also exist in MDE-based PLAs for DRE systems, so they are not limited solely to event-based DRE systems as described in our case study. In the remainder of this section, we first discuss the challenges and solutions associated with domain-specific component framework evolution and DSML evolution in Section 5.1, then the challenges and solutions associated with domain model evolution in Section 5.2.

5.1 Challenges Stemming from Capturing New Requirements into Existing MDE-based PLAs for DRE Systems

Context. Evolution is a natural occurrence in software development and an inevitable part of the software PLA lifecycle (Chapin et al., 2001). The changes may be initiated to correct, improve, or extend assets or products. Because assets are often dependent on other assets, changes to one asset may require corresponding changes in other assets. Moreover, changes to assets in PLAs can propagate to affect all products using these assets. A successful process for PLA evolution must therefore manage these changes effectively (McGregor, 2003).

Problem: New requirements impact metamodels and component frameworks. DRE systems must evolve to adapt to changing requirements and operational contexts such as supporting new features. In addition, when some emerging technologies become sufficiently mature, it is often desirable to integrate them into existing PLAs for DRE systems.

For example, in our Boeing Bold Stroke case study, depending on system requirements, different product variants in the Bold Stroke PLA may require different levels of QoS assurance for event communication, including timing constraints, event delivery latency, jitter, and scalability. Even within the same product variant, different levels of QoS assurance may be required for different communication paths, depending on system criticality (e.g., certain communication paths between components may require more stringent QoS requirements than others).

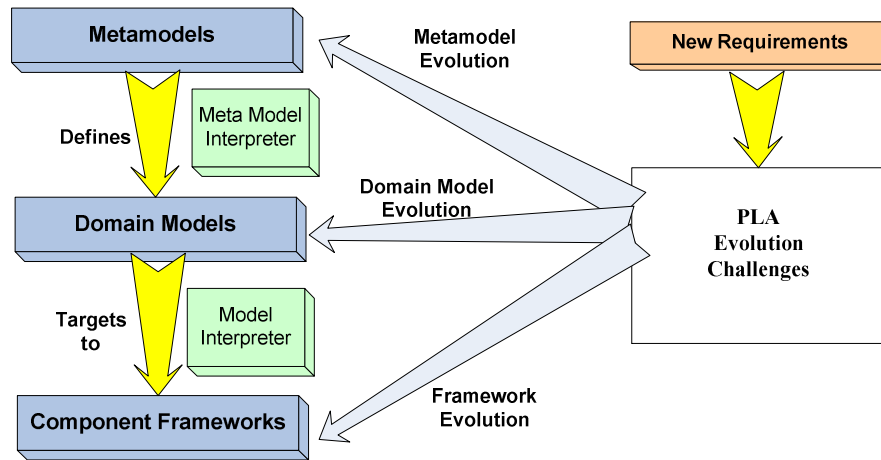


Figure 8. Challenges Stemming from Adding New Requirements into Model-Driven Software PLAs

Solution: Evolve a PLA systematically through framework and metamodel enhancement. A layered PLA can reduce software design complexity by separating concerns and enforcing boundaries between different layers. Because different layers in a PLA need to interact with each other through predefined interfaces, to integrate new requirements into a PLA, all layers must evolve in a systematic manner. This evolution can be generalized to the following three steps:

1. **Component framework evolution.** As discussed in Section 3, frameworks are often built atop middleware and OS platforms to provide the runtime environment of DRE systems. As a result, whenever a DRE system must evolve to adapt to new requirements, component frameworks are often affected because they have direct impact on the system.
2. **DSML evolution.** DSML metamodels and interpreters are often used to capture the *variability* and *features* of DRE systems to expose different capabilities for different product variants. As discussed in Section 3 and shown in Figure 2, typically the DSMLs for vertical application domains have a higher level of abstraction than DSMLs for horizontal platform domains. These lower level DSMLs are built atop domain-specific component frameworks and are often used to glue different component framework entities together to form a complete application. Therefore, the evolution of lower level DSMLs should be performed after framework evolution is completed.
3. **Domain model evolution.** The DSML metamodel defines a type system to which domain models must conform. Because the changes to the metamodel of a DSML often invalidate the existing domain models by re-defining the type system, domain model evolution must be performed after the DSML evolution.

In the remainder of this section, we further elaborate the solution approach and describe how it applies to our case study.

5.1.1 Component Framework Evolution

Component frameworks consist of a set of core reusable components that can be configured using well-defined interfaces. In order to capture the commonalities of software PLAs, one must formulate a set of usage patterns. The component frameworks encapsulate these usage patterns and provide reusable libraries that contain wrapper façades for the underlying implementation classes and shield component developers from tedious and error-prone programming tasks associated with lower-level details. Component frameworks are typically designed by analyzing various potential problems that the frameworks might address and identifying which parts of each solution are the same and which areas of each solution are unique through the SCV analysis.

The first step is to define the domains (i.e., the problem areas a framework addresses) and the context of the framework. The next step is to define the attributes that recur across all members of the family of products based on the framework. The final step is to describe the attributes unique to the different members of the family of products. The SCV analysis requires extensive knowledge about the domain and the PLA requirements so one can reason what parts of the system should be implemented by the framework (commonalities) and what parts of the system should be specialized in subclasses or parameters (variabilities). To implement such design usually requires effective and skillful use of programming language features, such as templates and virtual functions, in conjunction with design patterns (Gamma et. al. 1995).

Applying the solution to the EQAL case study. In our EQAL case study, the scope is to design a framework to simplify the event communication between the event sources and event sinks of PRISM components. The commonality in this scope is straightforward, i.e., every software product instance should implement an event-based publish/subscribe pattern. The variability of the EQAL Runtime Framework results from different concrete service types that provide different interfaces and different QoS mechanism for the event communication. Because different real-time publish/subscribe services depend on different representations of real-time QoS properties, the EQAL Runtime Framework implements the adapter pattern that converts a service-independent representation of real-time properties into a service-specific representation.

The benefits of EQAL’s design are twofold: (1) component developers need not concern themselves with peculiar configuration interfaces, and (2) no matter what changes occur to the underlying publish/subscribe services, the interface exposed to components does not change. The EQAL Runtime framework also implements the strategy pattern to enhance the extensibility by allowing new publish/subscribe services to be easily plugged-in. This design results in a pluggable publish/subscribe service implementation that is interchangeable and extensible, and enables all event communication mechanisms supported by EQAL to provide the same interface, yet can also be configured with different strategies and QoS configurations even facing the domain evolution of adding new publish/subscribe service types.

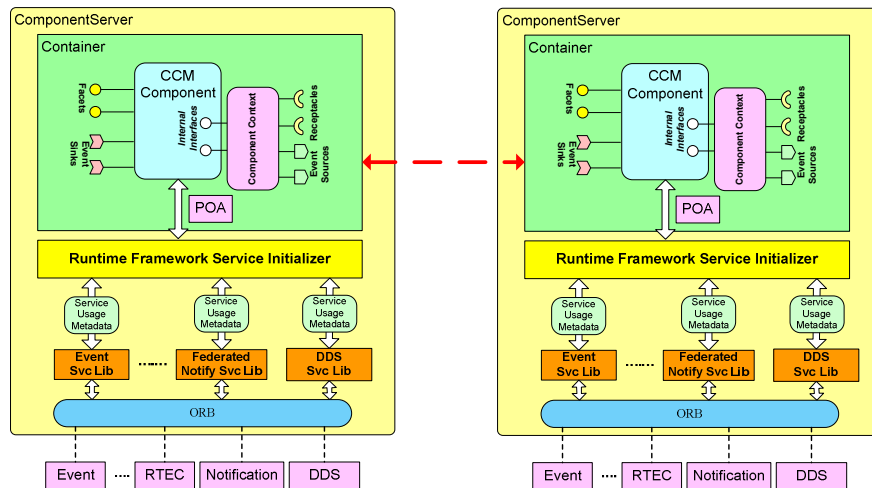


Figure 9. Component Framework Architecture of EQAL

5.1.2 DSML Evolution

The core component in the DSML is the metamodel. To help understand the context of domain evolution, Figure 10 presents a matrix of several evolution tasks that require automated assistance to manage the various dependencies among metamodels, instance models, and corresponding source code. As shown at the top of Figure 10, a metamodel represents a modeling language definition that is instantiated to represent end-user intentions in a specific domain. Elements in the instance models (middle of figure) have meta-types that are specified in the metamodel. A vertical transformation (i.e., a transformation that goes across abstraction layers) exists between the instance models and the legacy source code at the bottom, which represents updates that are needed in one artifact that are triggered by a change at a different layer of abstraction. Correspondingly, horizontal transformation occurs at the same layer of abstraction to address changing requirements (i.e., the Δ at each layer represents a horizontal transformation).

To simplify the evolution of DSMLs, the reusability of metamodels is crucial when the domain becomes complex. Ideally, a metamodel can be developed based on a set of reusable (usually smaller) metamodel units, with different units capturing different aspects in the domain of interest. For example, these metamodel units might include different variations of signal flow, finite state machines, data type specifications, and petri-nets. The unified metamodel can then extend these units and glue them together. This technique is called *compositional metamodeling* (Karsai et al., 2004), and the main motivation of this technique is to make metamodels more scalable and easier to evolve.

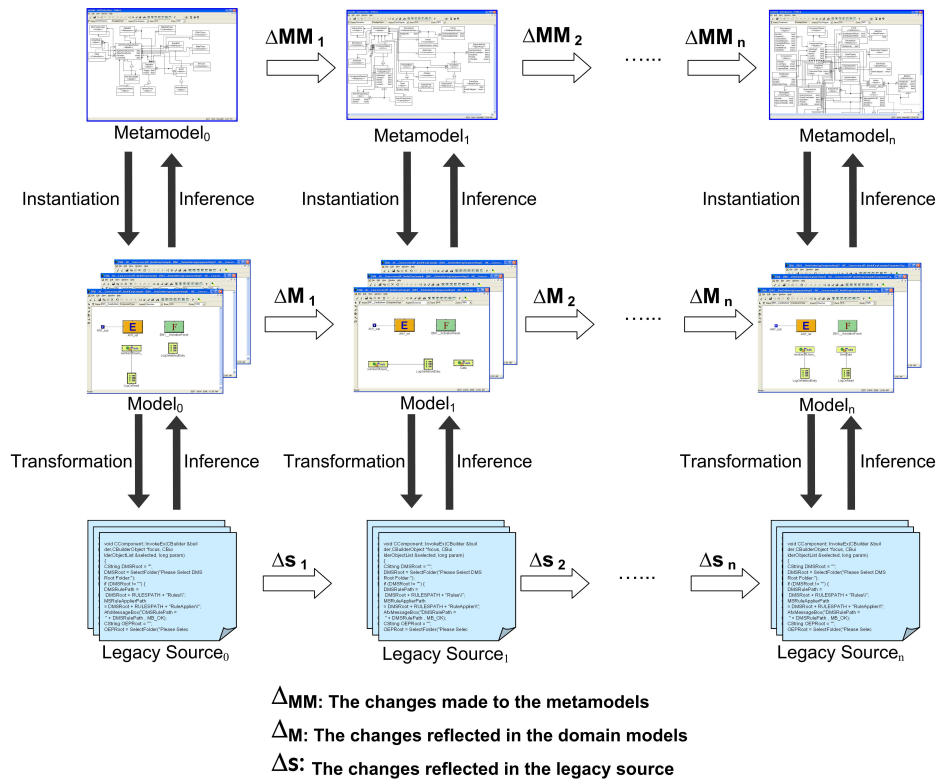


Figure 10. A Matrix of Evolution Activities within DSMLs

Composition metamodeling provides a capability for reusing and combining existing modeling languages and language concepts. When changes need to be made in the metamodel units to reflect a better understanding of the given aspect in the domain, such changes can be propagated automatically to the metamodels that utilize them. Furthermore, by precisely specifying the extension and composition rules, models specified in the original domain language can be automatically translated to comply with the new, extended and composed, modeling language. Another important benefit of compositional modeling is its ability to add new capabilities while simultaneously leveraging prior constraints and model generators of existing DSMLs, thus it is ideal for evolving existing DSMLs to address new requirements.

Applying the solution to the EQAL case study. EQAL is implemented within GME, which offers the compositional modeling capability. When new publish/subscribe services are integrated, a new DSML can be designed within GME and import the old EQAL metamodel as a reusable “library.” Apart from being read-only, all objects in the metamodel imported through the library are equivalent to objects created from scratch. Because the new publish/subscribe services share much commonality between the existing publish/subscribe services that EQAL already supports, when the old EQAL metamodel is imported as a library, subtypes can be created and instances from the metamodel library can refer to library objects through references.

5.2 Challenges Stemming from Migrating Existing Domain Models with MDE-based PLA Evolution

Context. The primary value of the MDE paradigm stems from the models created using the DSML. These models specify the system from which the executable application can be generated or composed. Changes to the system can be modeled and the resulting executable model is thus a working version of the actual system. Unfortunately, if the metamodel is changed, all models that were defined using that metamodel may require maintenance to adapt to the semantics that represent the system correctly. Without ensuring the correctness of the domain models after a change to the domain, the benefits of MDE will be lost. The only way to use instance models based on the original metamodel is to migrate them to use the modified metamodel. During this migration process, we must preserve the existing set of domain model assets and allow new features to be added into domain models; ideally, with as little human intervention as possible.

Problem: Existing domain model evolution techniques require excessive human intervention. As illustrated in Figure 11, to preserve the existing set of domain model assets, old domain models must be transformed to

become compliant with the changed metamodel. In the MDE research community, particularly in the DSML community, research has been conducted on using model transformation to address metamodel evolution (Sprinkle et al., 2003, Gray et al., 2006, Jouault 2006). The underlying structure of models, particularly visual models, can be described by graphs. Model transformation research has therefore often been conducted in the context of graph transformation. In particular, recent research (Vizhanyo et al., 2004, Balogh et al. 2007) has shown that graph transformation is a promising formalism to specify model transformation rules.

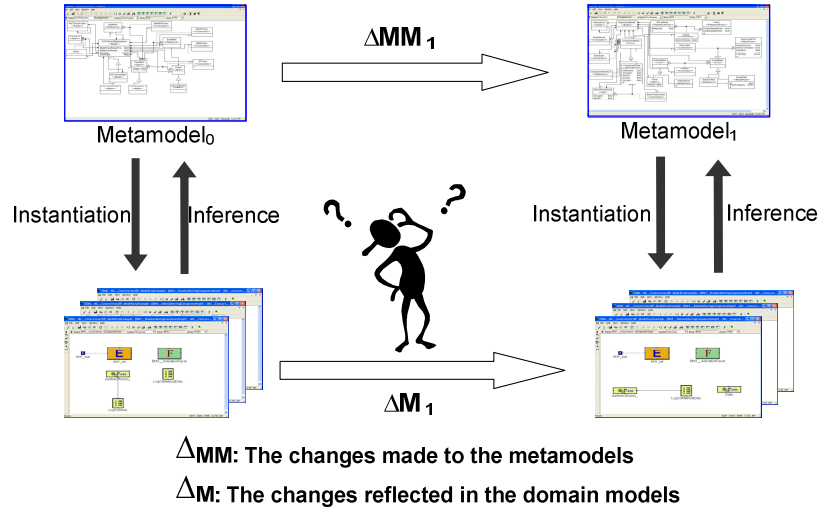


Figure 11. Domain Model Evolution Problem

Most existing model transformation techniques, however, require the transformation be performed *after* the domain metamodel has changed. For example, when an old metamodel is modified and a new metamodel-based on it is created, the model transformation must consider both the old metamodel and new metamodel as input, and then manually specify the model transformation rules based on these two metamodels by using a transformation specification language provided by the transformation tool. Although such a design approach could solve the model transformation problem, it introduces additional effort in specifying the model transformation rules, even if the metamodel evolution is minor (e.g., a simple rename of a concept in the metamodel). This additional effort is particularly high when the metamodels are complex, because the transformation tool must take both complex metamodels as input to specify the transformation.

Solution: Tool-supported domain model migration. To preserve the assets of domain models, our approach is to integrate *model migration* capabilities into the metamodeling environment itself. This approach is sufficiently generic to be applied to any existing metamodeling environment. A description of the change in semantics between an old and a new DSML is a sufficient specification to transform domain models such that they are correct in the new DSML. Moreover, the pattern that specifies the proper model migration is driven by the change in semantics and may be fully specified by a model composed of entities from the old and new metamodels, along with directions for their modification.

Below we describe how syntactic- and semantic-based model transformation approaches can be integrated to address the domain model migration problem.

5.2.1 Integration of Syntactic-based and Semantic-based Domain Model Evolution

Based on the characteristics of metamodel change, researchers have shown that 14 atomic types of metamodel changes can be defined (Sprinkle et al., 2004), as shown in Table 1. These results provide intuition into the problem of domain model evolution. In some cases, the semantics can be easily specified. For example, if the metamodel designer deletes an atom called “foo” in the metamodel and creates a new atom called “bar” we can then specify the semantics of the change as:

```
replace(Atom("foo") -> Atom("bar"));
```

<i>Type of metamodel change</i>		<i>Affected domain models (of this type) are present</i>	<i>Change is required</i>		<i>Equivalent schema change (for database)</i>
<i>Additions</i>					
<i>1</i>	Addition of new type A	<input type="checkbox"/>	<input type="checkbox"/>		Addition of table A
<i>2</i>	Addition of new attribute of type A	<input checked="" type="checkbox"/>	<input type="checkbox"/>		Addition of column in existing table A
<i>3</i>	Addition of association between types B and C	<input checked="" type="checkbox"/>	<input type="checkbox"/>		Addition of column for database key reference between two tables B and C
<i>4</i>	Addition of type(s) E derived from type D	<input checked="" type="checkbox"/>	<input type="checkbox"/>		Creation of view E based on existing table D
<i>5</i>	Addition of constraint on type F	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		Addition of database constraint F
<i>Deletions</i>					
<i>6</i>	Deletion of an attribute of type A	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		Deletion of column in non-empty table A
<i>7</i>	Deletion of an existing type B	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		Deletion of non-empty table A
<i>8</i>	Deletion of association between types D and E	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		Deletion/Rename of database key in table D which is used in table E
<i>9</i>	Deletion of constraint on type F	<input checked="" type="checkbox"/>	<input type="checkbox"/>		Removal of database constraint
<i>Modifications</i>					
<i>10</i>	Renaming type A	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Rename non-empty table A
<i>11</i>	Renaming attribute of type A	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Rename column in non-empty table A
<i>12</i>	Changing type of B	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		Redefinition of view B
<i>13</i>	Addition of type(s) E derived from type D , that replaces D in a certain context(s)	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		Creating a new view E that some stored procedures will refer to instead of D
<i>14</i>	Modification of constraint on type F	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		Modification of database constraint F

Table 1: Changes that Require a Paradigm Shift (Sprinkle et al., 2003b)

Syntactic metamodel changes, however, can often affect semantic changes, which result in a highly challenging task in model migration, i.e., *semantic migration*. Semantic migration requires that the meaning of the old domain models be preserved after the transformation and that the new domain models conform to the entire set of static constraints required in the new domain.

For model migration, we generalized two approaches to perform model transformation with semantic migration. In the first approach, given two *distinct* metamodels, source metamodel and destination metamodel, we can perform a transformation that converts the source models in entirety to the destination models. This means that a complete set of rules is needed to convert each entity in the models. In the second approach, we create a *unified* metamodel (old + new), such that both old and new domain models are valid. Developers can then write transformation specifications that convert those parts of the model belonging to the source part of the paradigm to equivalent models in the destination part of the paradigm.

We have found that the second approach is much cleaner and user-friendly than the first approach because it requires much less human effort. In this second approach, after the unified metamodel is formulated, we use an “SQL-like” declarative language that allows one to query and change the model to define model transformation rules. The *Embedded Constraint Language* (ECL), used by the C-SAW GME plug-in (Gray, 2001), is such a language. ECL is a textual language for describing transformations on visual models. Similar to the Object Constraint Language (OCL) defined in OMG’s UML specification, ECL provides concepts such as collection and

model navigation. In addition, the ECL also provides a rich set of operators that are not found in the OCL to support model aggregations, connections, and transformations. ECL is a declarative language that allows one to specify the formal transformation rules of the syntax translator to capture the semantic migration.

In previous work, we showed how ECL can be used to accomplish several model transformation tasks (Gray et al., 2006). As an input language to C-SAW, ECL can support aspect modeling, as well as the ability to scale a base model to a larger model with replicated structures. Figure 12 illustrates an input source model being transformed by an ECL transformation rule to generate a new target model. An example of using ECL to handle the domain model migration in our case study is described in the next subsection.

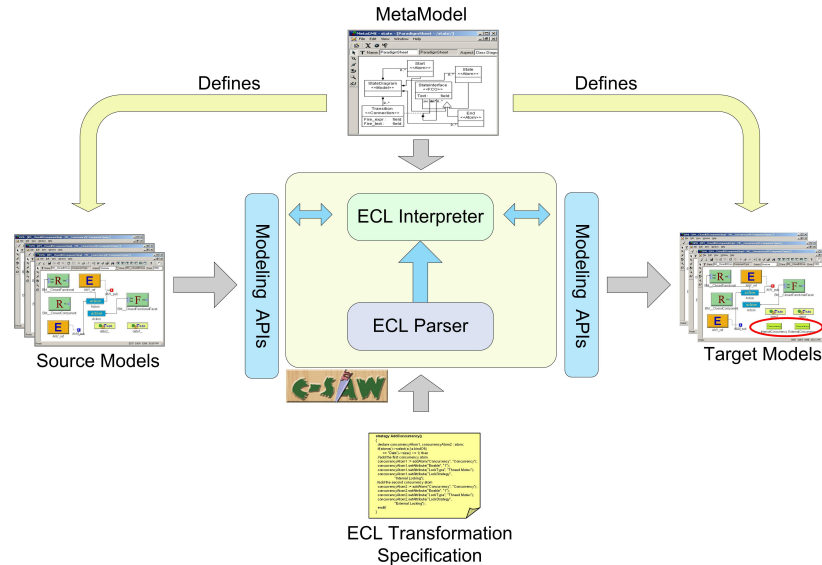


Figure 12. Model Transformation Using the ECL (Gray et al., 2006)

Applying the solution to EQAL case study. In an old version of the EQAL metamodel there is a modeling object type called “EventChannelGateway,” which can be used to federate different event channels together (Edwards et al., 2004). The definition of such a modeling element in a metamodel is similar to defining a class in C++ or Java. With domain evolution, this EventChannelGateway object type needs to be defined as an *abstract base* type (similar to the *abstract base* class concept in C++ or Java), and two new derived types called IIOPGateway and UDPGateway are defined in order to configure different underlying transport protocols between event channels. An issue arises regarding the type assignment of EventChannelGateway elements; depending on the context, these elements could be migrated to either the type of IIOPGateway or UDPGateway. In cases like these, it is quite challenging to discover the semantics of the change, i.e., the semantics of the model elements cannot be deduced from the syntax. To require that such algorithms provide actual semantic migration capabilities necessitates human input because semantic changes in metamodels cannot be captured through syntactic changes alone.

Figure 13 shows the BasicSP application scenario (Balasubramanian et. al. 2005b) in the Boeing Bold Stroke PLA. We use the BasicSP scenario as an example to showcase the problems encountered when evolving PLAs for component based DRE systems and motivate the need of ECL for model transformation. In this figure, two component instances named BMDevice and BMClosedED are connected with each other through a real-time event channel provided by TAO’s Real-time Event Service. An event channel consists of one RTEC_Proxy_Consumer module and RTEC_Proxy_Supplier module, which could be configured with various QoS settings, such as event dispatching threading models, priority configuration and periodic event processing configurations. Consider a domain evolution scenario, where the Real-time Event Service is not the desired choice for a particular Bold Stroke product variant, so it must be replaced with the TAO Federated Notification Service. In this case, the current domain model of Figure 13 will become invalid and must be migrated to the new EQAL DSML that supports the configuration of TAO’s Federated Notification Service.

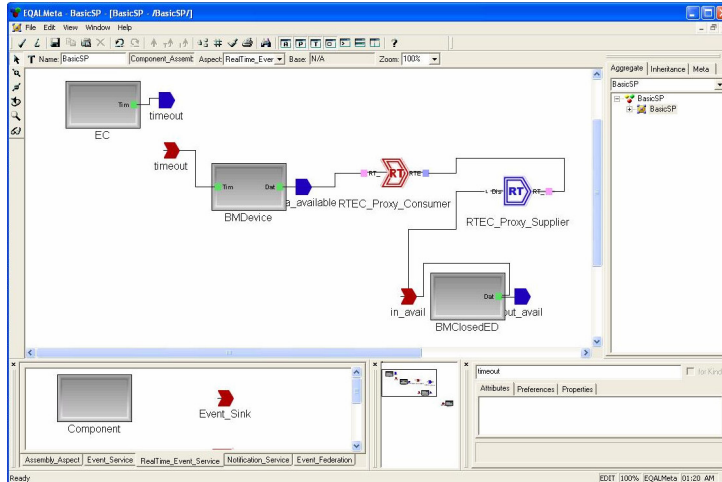


Figure 13. EQAL Configuring Real-time Event Service Between Two Components

With ECL, a model transformation rule can be defined to accomplish the model migration task noted above. In the ECL, a strategy represents a transformation rule that is applied to a specific location of a model. A query can be written in the ECL to define a collection of models that need to be transformed, and a strategy can be invoked on the collection. The strategy below specifies the desired model migration. The semantic meaning of this transformation is straightforward, i.e., line 1 declares the strategy based on the ECL syntax; lines 4-10 find the interested model elements and their associations that are based on TAO's Real-time Event Service; line 11 removes the found model elements, and lines 13-20 replace these model elements and associations with TAO's Federated Notification Service.

```

1. strategy ChangeToFNS() {
2.   declare FNS_Proxy_Consumer, FNS_Proxy_Supplier : model;
3.
4.   // Find interested model elements...
5.   if(atoms()->select(a | a.kindOf() = "RTEC_Proxy_Consumer")->size() >= 1) then
6.
7.     //get the RTEC_Proxy_Consumer model element
8.     //and its connections
9.     ...
10.    //delete the RTEC_Proxy_Consumer model element
11.    RTEC_Model.deleteModel("RTEC_Proxy_Consumer", "RTEC_proxy_consumer");
12.
13.    //add the FNS_Proxy_Consumer model
14.    FNS_Proxy_Consumer:= addModel("FNS_Proxy_Consumer", "FNS_proxy_consumer");
15.    FNS_Proxy_Consumer.setAttribute("Reactive", "1");
16.    FNS_Proxy_Consumer.setAttribute("LockType", "Thread Mutex");
17.
18.    //add the connections
19.    RTEC_Model.addConnection("Event_Source_Proxy_Consumer", event_source, FNS_Proxy_Consumer);
20.    RTEC_Model.addConnection("Proxy_Supplier_Event_Sink", FNS_Proxy_Consumer, event_sink);
21.
22.    //do similar to the FNS_Proxy_Supplier model
23.    ...
24.  endif;
25. }

```

6. Future Trends

This section discusses the future trends in the areas of MDE and component middleware, and how together they are impacting MDE-based PLAs, particularly for DRE systems.

Emerging Interest in Domain-Specific Modeling. The interest and adoption of DSLs over the past decade has surged. Strong support for basic research has been committed by the large European Union ModelWare and ModelPlex projects, which are funded at 30M Euros (ModelWare Project, 2006). Metamodeling tools that support DSM continue to emerge from both commercial and open source projects (e.g., Microsoft's DSL Toolkit (Microsoft, 2006) and the Eclipse Modeling Project (Eclipse, 2007)), as well as numerous academic research projects (e.g., Vanderbilt's Generic Modeling Environment (GME, 2007)). Initial success stories from industry adoption of DSM have been reported; the newly created DSM Forum (DSM Forum, 2007) serves as a repository

of several dozen successful projects (mostly from industry, such as Nokia, Dupont, Honeywell, and NASA) that have adopted DSM. Over the past five years, the annual DSM workshop at OOPSLA (42 international participants in 2006) provides a venue for reporting experiences in DSM research and practice.

Future trends of MDE tools for DRE systems. MDE has already played an important role in the assembly, configuration and deployment lifecycle stages of today's DRE systems. We envision next generation MDE tools will seamlessly integrate all lifecycle stages of software product lines, including requirement management, functionality specification, QoS specification, system partitioning and implementation, component assembly and packaging, system configuration, system planning and analysis and runtime system management. With such seamless integration, models will become vital artifacts in all aspects of software PLA development lifecycle, and sophisticated model transformation techniques will bridge the gap between models in different lifecycle stages. The need for seamless integration of models across the lifecycle is driving the need for integration across a collection of different modeling tools, where each offers some advanced capability not found in another tool. The need for tool integration will continue to heighten the role that model transformation plays as the key enabler of model sharing (Sendall et al., 2003).

Future trends of component middleware for DRE systems. The success of component middleware technologies has resulted in DRE systems created by customizing pre-existing COTS components rather than creating them from scratch. The increased use of pre-existing components shifts the focus from development to configuration and deployment of COTS components. With more COTS components provided by different vendors, the capability of *heterogeneous* deployment becomes a challenging task to evolve today's DRE systems.

Future component middleware technologies will enable rapid development of adaptive large scale DRE systems to accommodate changing operating environments. To facilitate the development of large-scale DRE systems, component middleware must support the agility in business service provisioning within and across organizations while ensuring the quality of service. The combination of these two techniques will finally enable software PLA developers to capture and represent adaptability of DRE systems at the business level and automatically translate this business adaptability into component and process adaptability.

As PLAs become more complex, they will be adopted into software-intensive systems of very large-scale, as typified by the focus of Ultra Large-Scale systems (Ultra Large-Scale, 2007). In such cases, it is not unrealistic to imagine the PLAs using multiple different middleware platforms. To accommodate these requirements demands new investigation in deployment and configuration across heterogeneous middleware platforms. This heterogeneity also adds to the challenges in provisioning QoS end-to-end for these PLAs. All these require novel modeling capabilities that can abstract away the heterogeneity.

We envision these future trends will greatly simplify the development of MDE-based PLAs and make next generation DRE systems more robust.

7. Concluding Remarks

Change is a natural and inevitable part of the software-intensive system lifecycle. The changes may be initiated to correct, improve, or extend assets or products. Since assets are often dependent on other assets, changes to one asset may require corresponding changes in other assets. Moreover, changes to assets in PLAs can propagate to affect all products using these assets.

To use MDE-based PLA technologies effectively in practice requires practical and scalable solutions to the *domain evolution problem*, which arises when existing PLAs are extended and/or refactored to handle unanticipated requirements or better satisfy existing requirements. For example, changing metamodels in a PLA often invalidates models based on previous versions of the metamodels. Although software developers can manually update their models and/or components developed with a previous metamodel to work with the new metamodel, this approach is clearly tedious, error-prone, and non-scalable. A successful process for PLA evolution must therefore manage these changes effectively.

To rectify these problems, this chapter describes a layered and compositional architecture to modularize system concerns and reduce the effort associated with domain evolution. This chapter illustrates via a case study how systematic evolution with three ordered steps can maintain the stability of domain evolution against MDE-based software PLAs, and how structural-based model transformations help reduce human effort by automatically transforming existing domain models based on metamodel-based rules.

The following is a summary of lessons learned from our experience in evolving product-lines using MDE tools:

- **DSMLs and Component Frameworks are Highly Synergistic** . An MDE approach expedites PLA development with the proper integration of DSMLs and component frameworks. The component frameworks help shield the complexities of the design and implementation of modeling tools, and decouple many aspects of concerns between the modeling tools and the executable systems. In our case study, if the publish/subscribe service type is the only missing or changing concern in the Boeing Bold Stroke PLA (which is typical in our case), little new application code must be written, yet the complexity of the generation tool remains manageable due to the limited number of well-defined configuration “hot spots” exposed by the underlying infrastructure. Likewise, when component deployment plans are incomplete or must change, the effort required is significantly less than starting from the raw component middleware without MDE tool support.
- **Declarative-based Model Transformation Alleviates Transformation Effort**. Structural-based model transformations help maintain the stability of domain evolution of MDE-based DRE systems by automatically migrating domain models. A declarative-based model transformation language like ECL is an ideal approach in such a case. The case study presented in this chapter highlights the ease of specification and the general flexibility provided by the transformation engine.
- **Testing and Debugging of Transformation Specification is Still Hard** . Transformation specifications, such as those used to specify the transformation strategy in this chapter, are written by humans and prone to error. To improve the robustness and reliability of model transformation, there is a need for testing and debugging support to assist in finding and correcting the errors in transformation specifications. Ongoing and future work on ECL focuses on the construction of testing and debugging utilities to ensure the correctness of ECL transformation specifications.

All software in this chapter can be downloaded our websites. The EQAL framework is shipped as part of the CIAO and is available at <http://download.dre.vanderbilt.edu>. The EQAL DSML is available at <http://www.dre.vanderbilt.edu/cosmic/>. C-SAW is available at: <http://www.cis.uab.edu/gray/Research/C-SAW/>

References

- Atkinson C., Bayer J., Bunse C., Kamsties E., Laitenberger O., Laqua R., Muthig D., Paech B., Wüst J. & Zettel J. (2002) *Component-based Product Line Engineering with UML*. Addison-Wesley, 2002.
- Balasubramanian K., Balasubramanian J., Parsons J., Gokhale A., & Schmidt D.C. (2005a). “A Platform-Independent Component Modeling Language for Distributed Real-time and Embedded Systems,” *Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium*, San Francisco, CA, March 2005, pp. 190-199.
- Balasubramanian K., Krishna A., Turkay E., Balasubramanian J., Parsons J., Gokhale A. & Schmidt D.C. (2005b), Applying Model-Driven Development to Distributed Real-time and Embedded Avionics Systems, the International Journal of Embedded Systems, special issue on Design and Verification of Real-Time Embedded Software, April 2005.
- Balasubramanian K., Gokhale A., Lin Y., Zhang J. & Gray J. (2006). “Weaving Deployment Aspects into Domain-Specific Models,” *International Journal on Software Engineering and Knowledge Engineering*, June 2006, vol. 16., no. 3, pp. 403-424.
- Balogh A. & Varro D., "The Model Transformation of the VIATRA2 Framework," *Science of Computer Programming (Special Issue on Model Transformation)*, Fall, 2007.
- Batory D. (2006). “Multilevel Models in Model-Driven Engineering, Product lines, and Metaprogramming,” *IBM Systems Journal*, vol. 45, no. 3, 2006, pp.527-540.
- Bézivin, J. (2005). “On the Unification Power of Models,” *Software and Systems Modeling*, vol. 4, no. 2, pp. 171-188.
- Bollella G., Gosling J., Brosgol B., Dibble P., Furr S., Hardin D., & Turnbull M., *The Real-time Specification for Java*, Addison-Wesley, 2000.
- Buschmann, F., Meunier, R., Rohnert H., Sommerlad P., & Stal M. (1996). *Pattern-Oriented Software Architecture—A System of Patterns*. Wiley & Sons, 1996.
- Chapin N., Hale J., Kham K., Ramil J., & Tan W. (2001) “Types of Software Evolution and Software Maintenance,” *Journal of Software Maintenance: Research and Practice*, January 2001, pp. 3-30.
- Clements, P. & Northrop L. (2001). *Software Product-lines: Practices and Patterns*, Addison-Wesley.

- Coplien, J., Hoffman, D., & Weiss D. (1998), "Commonality and Variability in Software Engineering," *IEEE Software*, 15(6) November/December, 1998, pp. 37-45.
- Dashofy E.M., Hoek A. & Taylor R.N. (2002). "An Infrastructure for the Rapid Development of XML-Based Architecture Description Languages," *Proceedings of the 24th International Conference on Software Engineering*, Orlando, FL, May 2002, pp. 266-276.
- Deng G., Lenz G. & Schmidt D.C. (2005) "Addressing Domain Evolution Challenges for Model-driven Software Product-line Architectures (PLAs)," *Proceedings of the MoDELS 2005 Workshop on MDD for Software Product-lines: Fact or Fiction?*, Montego Bay, Jamaica, October 2005.
- DSM Forum. (2007). Domain-Specific Modeling Forum, <http://www.dsmforum.org/tools.html>
- Eclipse Modeling Project. (2007). <http://www.eclipse.org/modeling/>
- Edwards G., Deng G., Schmidt D.C., Gokhale A. & Natarajan B. (2004). "Model-driven Configuration and Deployment of Component Middleware Publisher/Subscriber Services," *Proceedings of the 3rd ACM International Conference on Generative Programming and Component Engineering*, Vancouver, CA, October 2004, pp. 337-360.
- Frankel D.S. (2003). *Model Driven Architecture: Applying MDA to Enterprise Computing*, John Wiley & Sons, 2003.
- Gamma E., Helm R., Johnson R. & Vlissides J. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- GME. (2007). Generic Modeling Environment, <http://escher.isis.vanderbilt.edu/downloads?tool=GME>
- Gore P., Schmidt D.C., Gill C. & Pyrali I. (2004). "The Design and Performance of a Real-time Notification Service," *Proceedings of the 10th Real-time Technology and Application Symposium*, Toronto, CA, May 2004, pg. 112.
- Gray J., Bapty T., Neema S. & Tuck J. (2001). "Handling Crosscutting Constraints in Domain-specific Modeling," *Communications of the ACM*, 44(10), pp. 87-93.
- Gray J., Lin Y., & Zhang J. (2006)., "Automating Change Evolution in Model-Driven Engineering," *IEEE Computer*, Special Issue on Model-Driven Engineering (Doug Schmidt, ed.), vol. 39, no. 2, February 2006, pp. 51-58.
- Gray J., Tolvanen J., Kelly S., Gokhale A., Neema S., & Sprinkle J. (2007)., "Domain-Specific Modeling," *CRC Handbook on Dynamic System Modeling*, (Paul Fishwick, ed.), CRC Press, 2007.
- Harrison T., Levine D., & Schmidt D.C. (1997), "The Design and Performance of a Real-time CORBA Event Service," *Proceedings of OOPSLA*, ACM, Atlanta, GA, October 1997, pp. 184-200.
- Hoek A., Mikic-Rakic M., Roshandel R. & Medvidovic N. (2001), "Taming Architectural Evolution," *Proceedings of the 8th European Software Engineering Conference (held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering)*, Vienna, Austria, September 2001, pp. 1-10.
- IBM (2007)., Rational Software, <http://www-306.ibm.com/software/rational/>
- iLogix (2006), Rhapsody, <http://www.ilogix.com/sublevel.aspx?id=284>
- Jouault F. & Kurtev I. (2006), On the Architectural Alignment of ATL and QVT. *Proceedings of ACM Symposium on Applied Computing*, Dijon, Bourgogne, France, (April 2006)
- Karsai G., Neema S., Abbott B., & Sharp D. (2002) "A Modeling Language and Its Supporting Tools for Avionics Systems," *Proceedings of 21st Digital Avionics Systems Conference*, August 2002. Volume 1, Issue , 2002 Page(s): 6A3-1 - 6A3-13 vol.1
- Karhinen A. & Kuusela J. (1998). "Structuring Design Decisions for Evolution," *Proceedings of Second International ESPRIT ARES Workshop*, Las Palmas de Gran Canaria, Spain, February 1998. Springer-Verlag, pp. 223-234.
- Karsai G., Maroti M., Lédeczi A., Gray J., & Sztipanovits J. (2004) "Composition and Cloning in Modeling and Metamodeling," *IEEE Transactions on Control System Technology* (special issue on Computer Automated Multi-Paradigm Modeling), vol. 12, no. 2, March 2004, pp. 263-278.
- Klusener S., Laemmel R. & Verhoef C. (2005), "Architectural Modifications to Deployed Software," *Science of Computer Programming*, vol 54, 2005, pp. 143--211.
- Krueger C.W. (2002). "Variation Management for Software Production Lines," *Proceedings of Second International Conference of Software Product Lines*, SPLC 2, San Diego, CA, August 2002, pp. 37-48.

- Lédeczi Á., Nordstrom, G., Karsai, G., Volgyesi, P., & Maroti, M. (2001). "On Metamodel Composition," *Proceedings of the 2001 IEEE international conference on control applications*, Mexico City, Mexico, pp. 756–760.
- Long E., Misra A., Sztipanovits J. (1998), "Increasing Productivity at Saturn," *IEEE Computer*, Volume 31, Issue 8, August 1998, pp. 35-43.
- Macala, RR., Stuckey, L., & Gross D. (1996). "Managing Domain-Specific, Product-Line Development," *IEEE Software*, Vol.14, No. 13, May 1996, pp. 57-67.
- Mens, T. and D'Hondt, T. 2000. "Automating Support for Software Evolution in UML," *Automated Software Engineering*. 7, 1, Mar. 2000, pp. 39-59.
- McGregor J.D. (2003), "The Evolution of Product-line Assets," Technical Report, *CMU/SEI-2003-TR-005m ESC-TR-2003-005*
- Microsoft Corporation. (2000), Microsoft COM Technologies DCOM, 2000.
- Microsoft Corporation. (2006). Microsoft Domain-Specific Language (DSL) Tools: Visual Studio 2005 Team System. <http://msdn.microsoft.com/vstudio/teamsystem/workshop/DSLTools>
- Microsoft Corporation. (2007), "Windows Embedded CE 6.0", <http://www.microsoft.com/windows/embedded/>
- ModelWare Project (European Union, 2006), <http://www.modelware-ist.org/>
- Ommering R., Linden F., Kramer J. & Magee J. (2002), "The Koala Component Model for Consumer Electronics Software," *IEEE Computer*, Volume 33 , Issue 3, March 2000, pp. 78–85.
- OMG. (2002), "MOF 2.0 Query / Views / Transformations RFP", OMG document ad/2002-04-10, April 2002
- OMG. (2003), "The Common Object Request Broker: Architecture and Specification," July 2003.
- OMG., (2004), "Data Distribution Service", OMG document, formal/04-12-02, Dec. 2004
- OMG. (2004b), "Light-Weight CORBA Component Model," OMG document, ptc/04-06-10
- OMG., (2005), "Real-time CORBA Specification," OMG document, formal/05-01-04, Jan. 2005
- OMG., (2005b), "MOF QVT Final Adopted Specification", OMG document, ptc/05-11-01.pdf, Nov., 2005
- OMG. (2006), "CORBA Component Model," OMG Document formal/2006-04-01 ed., April 2006.
- Ran A. & Kuusela J. (1996). "Design Decision Trees," *Proceedings of the Eighth International Workshop on Software Specification and Design*, March 1996, pg. 172.
- Roddick, J.F. (1992): Schema Evolution in Database Systems: An Annotated Bibliography. *SIGMOD Record* 21(4), 1992
- Roll W. (2003) "Towards Model-Based and CCM-Based Applications for Real-Time Systems," *Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, Hokkaido, Japan, May 2003, pp. 75-82.
- Schmidt D. C. (1993). "The ADAPTIVE Communication Environment: An object-oriented network programming toolkit for developing communication software," *Concurrency: Practice and Experience*, vol. 5, no. 4, , 1993, pp. 269–286.
- Schmidt D.C., Levine D. & Mungee S. (1998). "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, April 1998, pp. 294–324.
- Schmidt D.C., Stal M., Rohert H., & Buschmann F, (2000) *Pattern-Oriented Software Architecture: Concurrent and Networked Objects*, John Wiley and Sons, 2000.
- Schmidt D. C. (2002). "Middleware for real-time and embedded systems," *Communications of the ACM*, vol. 45, June 2002.
- Schmidt, D.C. (2006). "Model-Driven Engineering," *IEEE Computer*, February 2006, pp. 25-32.
- Schulte, M. (2003), "Model-Based Integration of Reusable Component-Based Avionics System," *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, Seattle, Washington, May 2005.
- Sendall, S. & Kozaczynski, W. (2003). "Model Transformation – the Heart and Soul of Model-Driven Software Development," *IEEE Software*, vol. 20, no.5, September/October 2003, pp. 42-45.

- Sharp D. (1999). "Avionics Product Line Software Architecture Flow Policies," *Proceedings of the 18th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, St Louis, MO, October 1999.
- SIEMENS (2007). <http://www.us.sbt.siemens.com/bau/products/default.asp>
- Sprinkle J., Agrawal A., Levendovszky T., Shi F., & Karsai G. (2003). "Domain Model Translation Using Graph Transformations," *Engineering of Computer-Based Systems*, Huntsville, AL, April 2003, pp. 159-167.
- Sprinkle J. (2003b) "Metamodel Driven Model Migration", PhD Dissertation, Vanderbilt University, Nashville, TN 37203, Aug., 2003.
- Sprinkle J. & Karsai G. (2004) "A Domain-Specific Visual Language for Domain Model Evolution," *Journal of Visual Languages and Computing*, vol. 15, no. 3-4, June 2004, pp. 291-307.
- Sun Microsystems. (2000). Java Remote Method Invocation Specification, revision 1.5, JDK 1.2, Oct. 1998.
- Sun Microsystems. (2001). Enterprise JavaBeans Specification. java.sun.com/products/ejb/docs.html.
- Szyperski C. (2002), *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, 2002.
- Timesys. (2002), Predictable Performance for Dynamic Load and Overload, 2002, www.timesys.com/prodserv/whitepaper/Predictable_Performance_1_0.pdf
- Tolvanen (2005) Tolvanen J.P. & Kelly S., Defining Domain-Specific Modeling Languages to Automate Product Derivation: Collected Experiences. *Proceeding of 9th Software Product Line Conference*, Rennes, France, 2005, pp. 198-209.
- Ultra Large-Scale. (2007). Ultra Large-Scale Systems: The Report. <http://www.sei.cmu.edu/uls/>
- Vizhanyo A., Agrawal A. & Shi F. (2004). "Towards Generation of Efficient Transformations," *Proceeding of ACM International Conference on Generative Programming and Component Engineering*, Vancouver, Canada, October 2004, pp. 298-316.
- White J., Schmidt D.C., & Gokhale A. (2005). "Simplifying Autonomic Enterprise Java Bean Applications via Model-driven Development: a Case Study," *Proceedings of 8th International Conference on Model Driven Engineering Languages and Systems*, Montego Bay, Jamaica, October 2005, pp. 601-615.
- Wind River Systems. (1998) "VxWorks 5.3," ww.wrs.com/products/html/vxworks.html
- Zandy V. C. & Miller B. P (2002). "Reliable network connections," *Proceedings of the Eighth Annual International Conference on Mobile Computing and Networking*, September 2002, pp. 95-106.