Addressing Domain Evolution Challenges in Model-Driven Software Product-line Architectures

Gan Deng
Department of EECS
Vanderbilt University
Nashville, TN 37203, USA
gan.deng@vanderbilt.edu

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

Douglas C. Schmidt
Department of EECS
Vanderbilt University
Nashville, TN 37203, USA
d.chmidt@vanderbilt.edu

ABSTRACT

It is hard to develop and evolve software product-line architectures (PLAs) for large-scale distributed real-time and embedded (DRE) systems. Although certain challenges of PLAs can be addressed by combining model-driven development (MDD) techniques with component frameworks, domain evolution problems remain largely unresolved. In particular, extending or refactoring existing software product-lines to handle unanticipated requirements or better satisfy current requirements requires significant effort. This paper describes techniques for minimizing such impacts on MDD-based PLAs for DRE systems through a case study that shows how a layered architecture and model-to-model transformation tool support can reduce the effort of PLA evolution.

Keywords

Model-driven development, Product-line Architectures, Model Transformation

1. Introduction

Software *product-line architectures* (PLAs) [20] are a promising technology for industrializing software development by focusing on the automated assembly and customization of domain-specific components, rather than (re)programming systems manually. Conventional PLAs consist of *component frameworks* [29] 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 *common/variability analysis* (CVA) [23], which captures key characteristics of software product-lines, including (1) *scope*, which defines the domains and context of the PLA, (2) *commonalities*, which describe the attributes that recur across all members of the family of products, and (3) *variabilities*, which describe the attributes unique to the different members of the family of products.

Despite improvements in third-generation programming languages (such as Java, C#, and C++) and runtime platforms (such as component and web services middleware), the levels of abstraction at which PLAs are developed today remains low relative to the concepts and concerns within the application domains themselves. A promising means to address this problem involves developing PLAs using *model-driven development* (MDD) [9] tools. As shown in Figure 1, MDD tools help raise the level of abstraction and narrow the gap between problem and solution domain by combining (1) metamodeling and model interpreters to create domain-specific modeling languages (DSMLs) with (2) CVA and object-oriented extensibility capabilities to create domain-specific component frameworks. DSMLs help automate repetitive tasks that must be accomplished for each product instance, e.g., generating code to glue components together or synthesizing deployment artifacts for middleware platforms Domain-specific component frameworks factor out common usage patterns in a domain into reusable platforms, which help reduce the complexity of designing DSMLs by simplifying the code generated by their associated model interpreters.

To use MDD-based PLA technologies effectively in practice, however, requires practical and scalable solutions to the *domain evolution problem* [30], which arises when existing PLAs are extended and/or refactored to handle unanticipated requirements or better satisfy current requirements. Although PLAs can be enhanced by combining component frameworks with DSMLs, existing MDD tools do not handle the domain evolution problem effectively since they require significant manual changes to existing component frameworks and metamodels. For example, changing metamodels in a PLA typically invalidates models based on previous versions of the metamodels. While 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.

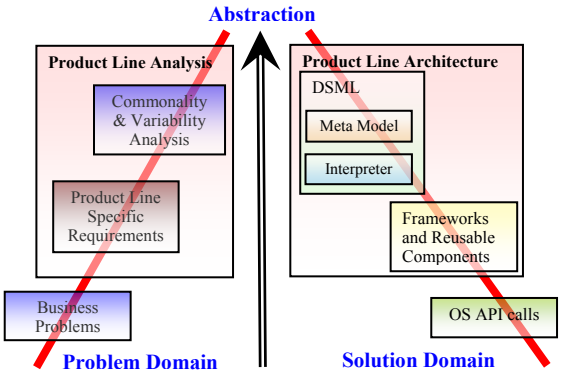

Figure 1: Using DSMLs and Component Middleware to Enhance Abstraction and Narrow the Gap between Problem and Solution Domain

This paper describes our approach to PLA domain evolution. We use a case study of a representative MDD-based tool for DRE system to describe how to evolve PLAs systematically and minimize human intervention for specifying model-to-model transformation rules as a result of metamodel changes. Our approach automates many tedious, time consuming, and error-prone tasks of model-to-model transformation to reduce the complexity of PLA evolution significantly.

The remainder of this paper is organized as follows: Section 2 describes our vision of the architecture of PLA for DRE systems, and introduces our case study, which applies the *Event QoS Aspect Language* (EQAL) MDD tool to simplify the integration and interoperability of diverse publish/subscribe mechanisms in the Bold Stroke PLA; Section 3 describes challenges we faced when evolving models developed using EQAL and presents our solutions to these challenges; Section 4 compares our work on EQAL with related research; and Section 5 presents concluding remarks.

2. Overview of MDD-based PLA and Case Study

This section presents an overview of an MDD-based PLA for DRE systems, focusing on the design concepts, common patterns, and software architecture. We then describe the structure and functionality of EQAL.

2.1 Design Concepts of MDD-based PLAs for DRE Systems

The MDD-based design and composition approach for embedded systems in [10] describes the benefits of combining DSML and reusable component frameworks. We believe this approach also applies to the design of PLAs for large-scale DRE systems. Figure 2 illustrates the high-level design principle and overall architecture of an MDD-based PLA solution that exploits a *layered* and *compositional* architecture to modularize various design concerns for DRE systems.

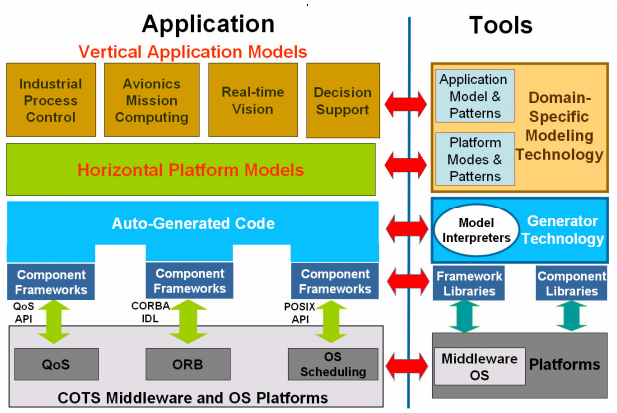

Figure 2. MDD-Based Product-line Architecture for DRE Systems

MDD-based PLAs for DRE systems are based on a core set of platforms, frameworks, languages, and tools. DRE systems increasingly run on commercial-off-the-shelf (COTS) middleware and OS platforms. Middleware platforms include *Real-time Java*, *Real-time CORBA*, *Real-time CCM*, and the *Data Distribution Service* (DDS) and OS platforms include *VxWorks*, *Timesys Linux*, and *Windows CE*. Since many DRE systems require a loosely-coupled distribution architecture to simplify extensibility, COTS middleware typically provides publish/subscribe-based communication mechanisms where application components communicate anonymously and asynchronously by defining three software roles*: publishers* generate events that are transmitted to *subscribers* via *event channels* that accept events from publishers and deliver events to subscribers. Event-based communication helps developers concentrate on the application-specific concerns of their DRE systems, and leaves the connection, communication, and QoS-related details to middleware developers and tools. An event-based communication model also helps reduce ownership costs since it defines clear boundaries between the components in the application, thereby reducing dependencies and maintenance costs associated with replacement, integration, and revalidation of components. Moreover, core components of event-based architectures can be reused, thereby reducing development, quality assurance, and evolution effort.

Component frameworks provide reusable building blocks of PLAs for DRE systems. These frameworks are increasingly built atop COTS middleware and OS platforms. Since the philosophy of COTS middleware and OS platforms catered to maintaining "generality, wide applicability, portability ad reusability," customized frameworks are often desired in DRE software product-lines to (1) raise the level of abstraction, and (2) offer product-line specific runtime environments. Examples of component frameworks include *domain-specific middleware services* layer in the Boeing Bold Stroke PLA [26], which supports many Boeing product variants, such as F/A-18E, F/A-18F, F-15E, and F-15K, using a component-based, publish/subscribe platform built atop *The ACE ORB* (TAO) [27] and *Prism* [28], which is QoS-enabled component middleware influenced by the *Lightweight CORBA Component Model* (CCM) [19]. The Boeing Bold Stoke PLA supports 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).

Domain-specific modeling languages (DSMLs) and patterns facilitate the model-based design, development, and analysis of *vertical application domains*, such as industrial process control, telecommunications, and avionics mission computing. Example DSMLs the include *Saturn Site Production Flow* (SSPF), which is a manufacturing execution system serving as an integral and enabling component of the business process for car manufacture industry [11] and the *Embedded System Modeling Language* (ESML) [12], which models mission computing applications in the Boeing Bold Stroke PLA. DSMLs are also applicable to *horizontal platform domains*, such as the domain of component middleware for DRE systems, which provide the infrastructure for many veritical application domains. Examples of DSMLs for horizontal platforms include *Platform Independent Component Modeling Language* (PICML) [13], which facilitates the development of QoS-enabled component-based DRE systems and J2EEML [17], which facilitates the development of EJB applications. 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.

As shown in Figure 2, MDD-based PLA defines a framework of components that adhere to a common architectural style 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, MDD technologies are used to model PLA features and glue components together, e.g., they could be utilized to synthesize deployment artifacts [13] for standard middleware platforms.

2.2 The Design of the EQAL MDD Tool

The *Event QoS Aspect Language* (EQAL) is an MDD tool designed to reduce certain aspects of component-based publish/subscribe PLA-based DRE systems, such as the Boeing Bold Stroke PLA described in Section 2.1. EQAL is implemented using the Generic Modeling Environment (GME) [5], which is a toolkit that supports the development of DSMLs. The EQAL DSML provides an integrated set of metamodels, model interpreters, and standards-based component middleware that allowing PLA developers to visually configure and deploy event-based communication mechanisms in DRE systems via models instead of programming them manually. EQAL is an example of a DSML 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.

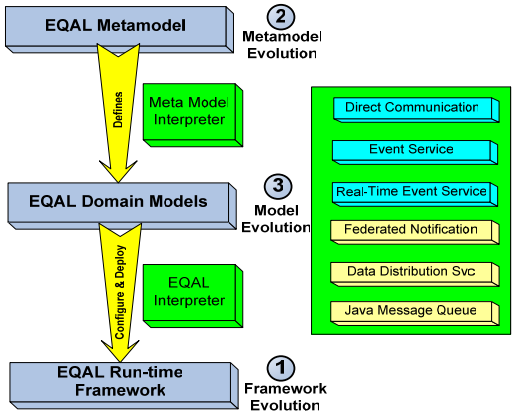

Figure 3. EQAL MDD Tool Architecture

As shown in Figure 3, EQAL is a layered architecture that supports several types of abstractions, which are subject to change stemming from domain evolution, as discussed in Section 3. The bottom layer is the *EQAL Runtime Framework*, which is a portable, OS-independent framework built atop the *Component-Integrated ACE ORB* (CIAO) QoS-enabled implementation of the Lightweight CCM specification. The EQAL

Runtime Framework provides an extensible way to deploy various event-based communication mechanisms, including a two-way event communication mechanism based on direct method invocation, the CORBA Event Service, and TAO's Real-time Event Service [24].

The middle layer in the EQAL architecture is a set of domain models that represent instances of modeled DRE systems. These models are created by 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 MDD architecture consists of metamodel that enables developers to model concepts of event-based DRE systems, including the configuration and deployment of various publish/subscribe services and how these services are used by CCM components. This layer also contains several model interpreters that synthesize various types of configuration files that specify QoS configurations, parameters, and constraints. The EQAL interpreters automatically generate publish/subscribe service configuration files and service property description files needed by the underlying EQAL Runtime Framework and CIAO middleware.

3. Resolving Challenges of MDD-based PLA when Facing Domain Evolution

This section examines challenges associated with evolving an MDD-based PLA in the context of the Boeing Bold Stroke PLA and the EQAL DSML. For each challenge, we explain the context in which the challenge arises, identify key problems that must be addressed, outline our approach for resolving the challenges, and describe how we can apply these solutions using EQAL.

3.1 Challenge 1: Capturing New Requirements into Existing MDD-based Software Product-lines for DRE Systems

Context. Change is a natural and inevitable part of the software PLA 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. A successful process for PLA evolution must therefore manage these changes effectively [15].

Problem → New requirements must be captured into existing PLAs. DRE systems must evolve to adapt to changing requirements and operational contexts. In addition, when some emerging technologies become sufficiently mature, it is often desirable to integrate them into existing PLAs for DRE systems. For example, depending on customer 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 other ones.

The event communication mechanisms currently supported by EQAL include: (1) two-way based event communication based on direct method invocation, (2) CORBA event service, and (3) TAO's Real-time Event Service [24]. Although the communication mechanisms provided by EQAL are applicable to many types of event-based systems, with the evolution in a domain and new technologies emerging, other event communication mechanisms may be needed. For example, TAO's reliable multicast *Federated Notification Service* is desired in certain DRE systems to address scalability and reliability. Likewise, the OMG's *Data Distribution Service* (DDS) [25] is often desired when low latency and advanced QoS capabilities are key product variant concerns. When these two new publish/subscribe technologies are added into the existing EQAL MDD tool, all layers in EQAL MDD architecture must change accordingly, including EQAL Runtime Framework, EQAL DSML and EQAL Domain Models. Moreover, since EQAL models have already been used in earlier incarnations of a PLA, such as Bold Stroke, we must minimize the effort required to migrate existing EQAL models to adhere to the new metamodels.

Solution → Evolve PLA *systematically* through framework and metamodel enhancement. A layered PLA can reduce software design complexity by separating concerns and enforcing boundaries between different layers. Since different layers in PLA still need to interact with each other through predefined interfaces, to integrate new requirements into a PLA, all layers must evolve in a *systematic* manner. As shown in Figure 3, for most PLAs for DRE systems we generalized this evolution to the following three ordered steps:

1. **Component framework evolution.** As discussed in Section 2.1, frameworks are often built atop middleware and OS platforms and provide the runtime environment to DRE systems. As a result, whenever the DRE systems must evolve to adapt to new requirements, component frameworks are often affected since 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 so a system can expose different capabilities for different product variants. Often, DSMLs are used to glue different component framework entities together to form a complete application, so typically DSML evolution 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. Since the changes to the metamodel of a DSML often invalidate the existing domain models by redefining the type system, domain model evolution must be performed after the DSML evolution.

We discuss the challenges and solutions associated with component framework and DSML evolution in the Section 3.1.1 and then discuss the challenges and solutions associated with domain model evolution in Section 3.1.2.

3.1.1 EQAL Framework Evolution

In our case study, the EQAL Runtime Framework provides a set of service configuration libraries that can configure various publish/subscribe services. Since these middleware services can be configured using well-defined and documented *interfaces*, we can formulate the usage patterns of such middleware services easily. The EQAL Runtime Framework can encapsulate these usage patterns and provide reusable libraries that (1) contain wrapper façades for the underlying publish/subscribe middleware services to shield component developers from tedious and error-prone programming tasks associated with initializing and configuring these publish/subscribe services and (2) expose interfaces to the external tools to manage the services, so that service configuration and deployment processes can be automated, as shown in Figure 3. I. To incorporate these new publish/subscribe technologies and minimize the impact on existing DRE systems, we used the *Adapter* and *Strategy* patterns so all event communication mechanisms supported by EQAL provide the same interface, yet can also be configured with different strategies and QoS configurations.

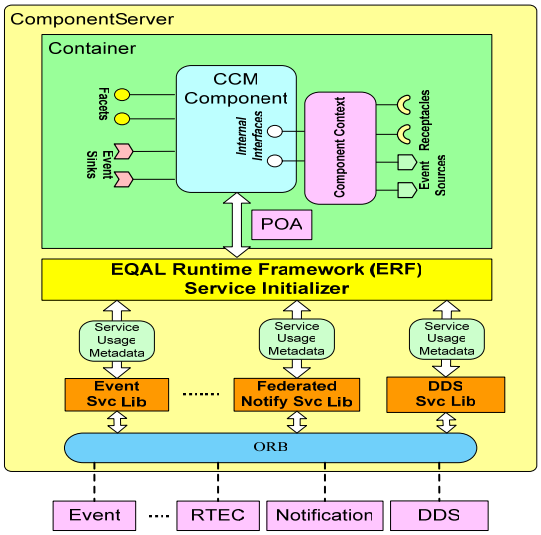

Figure 4. EQAL Runtime Framework Evolution

3.1.2 EQAL DSML Evolution

The EQAL metamodel must be enhanced to incorporate these new requirements, so system developers can model the behavior of new event-based communication mechanisms visually. For example, to enhance EQAL to support DDS and TAO's Federated Notification Service, the metamodel of the EQAL DSML must be changed. Since the EQAL metamodel defines the language to describe EQAL domain models, it is essential to minimize the impact on EQAL domain models, so that the EQAL domain models can be transformed easily to comply with the new EQAL metamodel.

Compositional metamodeling is a key idea to make metamodel scalable and easier to evolve. This technique provides a metamodel composition capability for reusing and combining existing modeling languages and language concepts. Since EQAL is implemented with GME, when new publish/subscribe services are integrated, we could design a new DSML within GME and import the old EQAL metamodel as a "library".. Apart from being read-only, all objects in the metamodel imported through the library are equivalent to objects created from scratch. Since the new publish/subscribe services share much commonality between the exiting publish/subscribe services that EQAL already supports, when the old EQAL metamodel is imported as library, we could create subtypes and in-

stances from the metamodel library and refer library objects through references..

3.2 Challenge 2: Migrating Existing Domain Models with MDD-based PLA Evolution

Context. The primary value of the MDD paradigm stems from the models created using the DSML. These models specify the system, and from the models the executable system can be generated or composed. Changes to the computer-based 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 computer-based system correctly. Without ensuring the correctness of the domain models after a change to the domain, the benefits of MDD 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 effort as possible.

Problem → Existing domain models evolution techniques require excessive human intervention. To address the challenge of preserving the existing set of domain model assets, old domain models must be transformed to become compliant with the changed metamodel. In the MDD research community, particularly in the DSML community, research has been conducted on using model transformation to address metamodel evolution. Since the underlying structure of models, especially visual models, can be described by graphs, most of the model transformation research has been conducted in the context of graph transformation. In particular, recent research [1,2] has shown that graph transformation is a promising formalism to specify model transformations rules.

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 designer must take both the old metamodel and new metamodel as input, and then manually specify the model transformation rules based on these two metamodels by using the "transformation behavior 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, since 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 bring *model migration* capabilities online, i.e., embed domain 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 and 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 [6].

3.2.1 Integration of Syntactic-based and Semantic-based Domain Model Migration

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

Table 1: Changes that Require a Paradigm Shift [6]

Based on the characteristics of metamodel change, researchers have shown that 14 "atomic" types of metamodel changes can be defined [6], as shown in Table 1. These results provide us the intuition into the problem. 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"));
```

Syntactic metamodel changes, however, can often affect semantic changes, which results in a highly challenging task in model migration, i.e., *semantic migration*. Semantic migration requires that the meaning of the old domain models is preserved after the transformation, and that the new domain models conform to the entire set of static constraints required in the new domain. In these cases, it is quite challenging to discover the semantics of the change. To make such algorithms provide actual "semantic migration" capabilities, human input will be necessary since semantic changes in metamodels can not be captured through syntactic changes alone.

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

It is evident that the second approach is much cleaner and user-friendly than the first approach since it requires much less human effort. We are therefore investigating the second model migration approach. In our approach, after the unified metamodel is formulated, we use an "SQL-like" declaratively 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 [2], seems to be a good candidate for such a language. The ECL is a textual language for describing transformations on visual models. Similar to the Object Constraint Language (OCL) defined in OMG's UML specification, the 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 imperative language that allows one to specify procedural style transformation rules of the syntax translator to capture the semantic migration.

3.2.2 EQAL Domain Model Evolution.

Figure 5 illustrates the BasicSP application scenario in the Boeing Bold Stroke PLA, in which two component instances named `BMDevice` and `BMClosedED` are connected with each other through 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. 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 TAO Federated Notification Service. In this case, the current domain model below will become invalid and must be migrated to the new EQAL DSML that supports the configuration of TAO's Federated Notification Service.

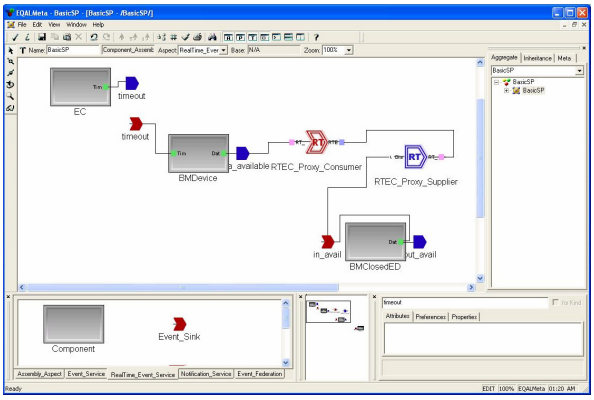

Figure 5. EQAL Configuring Real-time Event Service between Two Components

With ECL declarative language, we could create a model translator by defining strategies as below:

```
strategy ChangeToFNS() {

    declare FNS_Proxy_Consumer, FNS_Proxy_Supplier : model;

    // Find interested model elements…
    if(atoms()->select(a | a.kindOf() =
```

```
        "RTEC_Proxy_Consumer")->size() >= 1) then

    //get the RTEC_Proxy_Consumer model element
    //and its connections
       …
    //delete the RTEC_Proxy_Consumer model element
    RTEC_Model.deleteModel("RTEC_Proxy_Consumer",
                           "RTEC_proxy_consumer");

    //add the FNS_Proxy_Consumer model
    FNS_Proxy_Consumer:= addModel("FNS_Proxy_Consumer",
                                  "FNS_proxy_consumer");
    FNS_Proxy_Consumer.setAttribute("Reactive", "1");
    FNS_Proxy_Consumer.setAttribute("LockType",
                                    "Thread Mutex");

    //add the connections
    RTEC_Model.addConnection(
       "Event_Source_Proxy_Consumer",
       event_source,
       FNS_Proxy_Consumer);
    RTEC_Model.addConnection(
       "Proxy_Supplier_Event_Sink",
       FNS_Proxy_Consumer,
       event_sink);

    //do similar to the FNS_Proxy_Supplier model
    …
    endif;
}
```

The semantic meaning of this translator is straightforward, i.e., first find the interested model elements and their associations that are based on TAO's Real-time Event Service and replace these model elements and associations with TAO's Federated Notification Service.

4. Related Work

Software product-line is a viable software development paradigm that enables order-of-magnitude improvements in time to market, cost, productivity, quality, and other business drivers [20]. As MDD technology becomes more pervasive, there has been an increase in focus on technologies, architecture, and tools for applying MDD-based techniques into software PLAs. This section compares our research with related work.

Microsoft's Software Factory scheme [9] focuses on combining MDD- and component-based techniques with product-line principles to create highly extensible development tools quickly and cheaply for specific domains. The PLAs for DRE systems we describe in Section 2 are similar to the Software Factory scheme, but focuses on how aspects of PLAs for DRE systems should be designed and evolved throughout a system's lifecycle.

Generative software development techniques [19] develop software system families by synthesizing code and other artifacts from specifications written in textual or graphical domain-specific languages. Key concepts and idea in this paradigm include DSML, domain and application engineering. and generative domain models. Feature modeling [18] is a method and notation for capturing common/variable features in a system family. This software development paradigm is related to our approach, though in our MDD-based PLA we use domain-specific graphical DSML notations to describe the application semantics, instead of using a universal feature modeling notation since the latter is too restrictive for many DRE systems. .

Significant efforts have focused on evolution problems of model-based legacy systems. The *Atlas Transformation Language* (ATL) developed in the *Generative Model Transformer* project [22] aims to define and perform general transformations based on OMG's MDA technology. Atlas is a model transformation language specified both as a metamodel and as a textual concrete syntax, and a hybrid of declarative and imperative language. The *Graph Rewriting and Transformation* (GReAT) [21] tool provides a model transformation specification language to handle the model migration problem by explicitly defining complex graph patterns and pattern matching algorithms through models. While the methods mentioned above are powerful, they are also labor-intensive since transformations must be defined manually, which does not scale up for large-scale DRE systems. In contrast, our approach enables automatic transformation with limited human intervention that eliminates much of the tedious tasks of model evolution. C-SAW [2] is a general model transformation engine developed as a GME [5] plug-in and is compatible with any metamodel, i.e., it is domain-dependent and can be used with any modeling language defined within the GME. C-SAW, however, can only handle domain model transformations when the metamodel is not changed, while our approach can be used even when the metamodel has changed.

5. Concluding Remarks

Large-scale DRE systems are hard to build. Software product-line architectures (PLAs) are an important technology for meeting the growing demand for highly customized and reusable DRE systems. MDD-based PLA provides a promising means to develop software product-lines for DRE systems by combining metamodeling, DSMLs, interpreters, frameworks, and COTS middleware and OS platforms.

Software product-lines must inevitably evolve to meet new requirements. Adding new (particularly new unanticipated) requirements to MDD-based PLAs, however, often causes invasive modifications to the PLA's component frameworks and DSMLs to reflect these new requirements. Since these modifications significantly complicate PLA evolution efforts, they can outweigh the advantages of PLA development compared to one off development. To rectify these problems, a layered and compositional architecture is needed to modularize system concerns and reduce the effort associated with domain evolution. This paper illustrates via a case study how (1) structural-based model transformations help maintain the stability of domain evolution by automatically transforming domain models and (2) aspect-oriented model transformation and weaving helps reduce human effort by capturing model-based structural concerns.

References

[1] Jonathan Sprinkle, Aditya Agrawal, Tihamer Levendovszky, Feng Shi, Gabor Karsai, "Domain Model Translation Using Graph Transformations," *ECBS 2003*: 159-167

[2] Jeff Gray, Ted Bapty, Sandeep Neema, James Tuck, "Handling Crosscutting Constraints in Domain-specific Modeling," *Communicaton of ACM* 44(10): 87-93 (2001)

[3] Jayant Madhavan, Philip A. Bernstein, Erhard Rahm: "Generic Schema Matching with Cupid," *VLDB 2001*: 49-58, Roma, Italy

[4] Frank Budinsky, David Steinberg, Ed Merks, Ray Ellersick, Timothy Grose, "*Eclipse Modeling Framework*", Addison-Wesley 2004

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

[6] Jonathan Sprinkle, Gabor Karsai, "A Domain-Specific Visual Language for Domain Model Evolution", *Journal of Visual Language and Computation*, vol. 15, no. 3-4, pp. 291-307, Jun., 2004.

[7] Gan Deng, Jaiganesh Balasubramanian, William Otte, Douglas C. Schmidt, and Aniruddha Gokhale, "DAnCE: A QoS-enabled Component Deployment and Conguration Engine," *Proceedings of the 3rd Working Conference on Component Deployment*, Grenoble, France, November 28-29, 2005.

[8] Gan Deng, "Supporting Configuration and Deployment of Component-based DRE Systems Using Frameworks, Models, and Aspects," *OOPSLA '06 Companion*, October 2005, San Diego, CA, to appear

[9] Jack Greenfield, Keith Short, Steve Cook, Stuart Kent, John Crupi, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*, Wiley 2004

[10] Gabor Kasai, Janos Sztipanovits, Akos Ledeczi, and Ted Bapty, "Model-Integrated Development of Embedded software", Proceedings of the IEEE number 1, volume 91, Jan. 2003

[11] Karsai G., Sztipanovits J., Ledeczi A., Moore M., "Model-Integrated System Development: Models, Architecture and Process," *21st Annual International Computer Software and Application Conference (COMPSAC)*, pp. 176-181, Bethesda, MD, August, 1997

[12] http://www.isis.vanderbilt.edu/Projects/mobies/.

[13] Krishnakumar Balasubramanian, Jaiganesh Balasubramanian, Jeff Parsons, Aniruddha Gokhale, and Douglas C. Schmidt, "A Platform-Independent Component Modeling Language for Distributed Real-time and Embedded Systems," *Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium*, San Francisco, CA, March 2005

[14] George Edwards, Gan Deng, Douglas C. Schmidt, Anirudda Gokhale, and Balachandran Natarajan, "Model-driven Configuration and Deployment of Component Middleware Publisher/Subscriber Services," *Proceedings of the 3rd ACM International Conference on Generative Programming and Component Engineering*, Vancouver, CA, October 2004

[15] John D. McGregor, "The Evolution of Product-line Assets," Technical Report, *CMU/SEI-2003-TR-005m ESC-TR-2003-005*

[16] David Sharp. "Avionics Product-line Software Architecture Flow Policies," In *Proceedings of the Digital Avionics Systems Conference*, 1999

[17] Jules White, Douglas Schmidt, and Aniruddha Gokhale, "Simplifying Autonomic Enterprise Java Bean Applications via Model-driven Development: a Case Study", *Proceedings of ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems*, Montego Bay, Jamaica, October 5-7, 2005.

[18] Krzysztof Czarnecki, Simon. Helsen, and Ulrich. Eisenecker, "Staged configuration using feature models", In *Proceedings of the Third Software Product-Line Conference*, Robert Nord, 2004

[19] Krzysztof Czarnecki, Ulrich Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley 2000

[20] Paul Clements, Linda Northrop, *Software Product-lines: Practices and Patterns*, Addison-Wesley, ISBN 0201703327, August 20, 2001

[21] Aditya Agrawal, Gabor Karsai, Ákos Lédeczi, "An End-to-end Domain-driven Software Development Framework," *Proceeding of ACM SIGPLAN OOPSLA 2003 Domain Driven Design session ,* Anaheim, CA, 2003

[22] Available at Generative Model Transformer project website, http://www.eclipse.org/gmt/

[23] James Coplien, Daniel Hoffman, and David Weiss, "Commonality and Variability in Software Engineering" *IEEE Software*, 15(6) November/December, 37—45, 1998

[24] Tim Harrison and David Levine and Douglas C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service", *Proceedings of OOPSLA '97*, ACM, Atlanta, GA, October 6-7, 1997

[25] OMG's "Data Distribution Service for Real-time Systems Specification", version 1.0, Dec.2004. http://www.omg.org/docs/formal/04-12-02.pdf

[26] David Sharp and Wendy Roll, "Model-Based Integration of Reusable Component-Based Avionics System," in *Proceedings of the Workshop on Model-Driven Embedded Systems in RTAS 2003*, May 2003

[27] Douglas Schmidt, David Levine, and Sumedh Mungee, "The Design and Performance of Real-Time Object Request Brokers", *Computer Communications*, vol. 21, pp. 294–324, Apr. 1998

[28] Wendy Roll, "Towards Model-Based and CCM-Based Applications for Real-Time Systems," in *Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, Hokkaido, Japan, IEEE/IFIP, May 2003

[29] Clemens Szyperski, "*Component Software: Beyond Object-Oriented Programming*", Addison-Wesley, Dec. 1997

[30] Randall R. Macala, Lynn D. Stuckey, Jr. David C. Gross, "Managing Domain-Specific, Product-Line Development", *IEEE Software*, Vol.14, No. 13, May 1996