Simplifying the Development of Product-line Customization Tools via Model Driven Development

Jules White and Douglas Schmidt
{jules, schmidt}@dre.vanderbilt.edu
Department of Electrical Engineering and Computer Science,
Vanderbilt University, Nashville, USA

Abstract

Product-line architectures (PLA)s are a paradigm for developing software families by customizing and composing reusable artifacts, rather than handcrafting software from scratch. A promising way to reduce the effort of developing software PLAs and product variants is to leverage Meta-configurable Modeling Environment (MME) to provide domain-specific modeling tools for customizing the PLA. Since creating a group of DSML tools to customize a PLA is hard, however, it is useful to provide a toolchain that domain experts can use to evolve the DSML tools as the PLA's requirements or knowledge of the domain evolves. This paper presents two contributions to the study of constructing DSML tools for PLAs: (1) the Generic Eclipse Modeling System (GEMS), which is a meta-configurable modeling environmentl that enables the rapid construction of domain-specific modeling languages), and (2) the Model EXTension Language (MEXTL), which provides a modeling language and code generation capabilities to build layers of abstraction atop existing models.

Key Words: Model Driven Development, Product-line Architectures, Metamodeling, Domain-specific Modeling Languages

1 Introduction

Product-line architectures (PLAs)[1] offer developers the ability to produce a group of software packages rapidly that are targeted for different requirement sets by leveraging a common set of capabilities, patterns, and architectural styles. The design of a PLA is typically guided by *scope, commonality, and variability* (SCV) analysis [6]. SCV 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.

After constructing a PLA, product-line engineers identify each product variant that must be produced from the architecture and its requirements. Once a PLA variant's requirements are obtained, these requirements must be mapped to the variabilities in the PLA. Traditional manual processes of mapping the requirements to the PLA involve software developers taking each requirement and attempting to determine the software components that must be in the variant, the components that must be configured, how each component must be changed, and how the components must be composed. Such manual approaches are tedious and error-prone and are a significant source of system downtime [2] [12].

An alternate approach to mapping the variant's requirements to the PLA involves the use of *Model-driven development* (MDD) [3] techniques and tools, which are designed to reduce the effort needed to capture system requirements and map them to the underlying PLA infrastructure. Models of PLA variants developed with MDD tools can be constructed and checked for correctness (semi-)automatically to ensure that application designs meet their requirements. MDD tools can also be used to generate the customization, composition, packaging, and deployment code to implement PLA variants [4] [5].

To reduce the effort of developing software PLAs and product variants, it is common to build DSML tools on top of a *Meta-configurable Modeling Environment* (MME). Creating a group of DSML tools to customize a PLA is, in itself, a challenging endeavor. Crucial to creating a viable MME to develop DSML tools is providing a platform that can provide (1) the appropriate level of abstract for each domain expert and (2) the means to evolve the DSML tools in response to changes in the PLA's requirements or knowledge of the domain.

To explore characteristics of MDD-based PLAs, we have developed an Enterprise Java Beans (EJB)-based constraints optimization system (CONST) that schedules pickup requests to vehicles, which is shown in Figure 1.

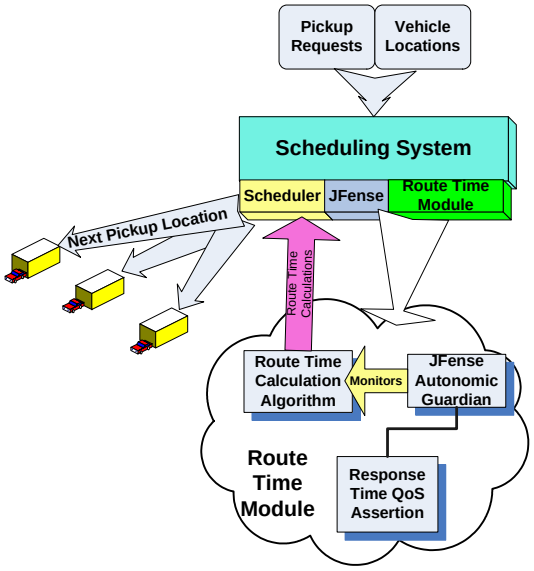

Figure 1. A Multi-Layered Autonomic Architecture for Scheduling Highway Freight Shipments

As shown in Figure 1, CONST manages a list of items that must be scheduled for pickup, a list of times that the items must arrive by, and a list of vehicles and drivers that are available to perform the pickup. It uses a constraint-optimization engine to find a cost effective assignment of drivers and trucks to pickups. CONST's optimization engine could be used schedule a wide variety of shipment types. In one configuration, for example, the system could schedule limousines to customers requiring a ride, whereas in another configuration the system could dispatch trucks to highway freight shipments. The CONST's optimization engine must therefore be customizable at design-time to handle these various domains effectively.

CONST must also be customizable at run-time to adapt to changing operating conditions. During peak traffic times, for instance, its optimization engine may need to use traffic-aware routing algorithms, whereas during off-peak times, it may switch to faster traffic-unaware algorithms. CONST also needs to handle failures differently depending on the target domain. For scheduling limousines to pickups, for example, a degradation of the time required to schedule a reservation below a threshold may require CONST's constraint engine to adapt to improve performance. When scheduling highway freight shipments, however, the threshold may be higher since pickup and drop-off windows are more flexible.

To support the degree of customization described above, we developed CONST as a PLA using SCV analysis, as follows:

- The **scope** is the constraint optimization system architecture and the associated components that address the domain of scheduling shipments to vehicles, including computing route times between vehicles and shipments, maintaining a list of waiting shipments, and calculating the cost of assigning a specific vehicle to a shipment.
- The **commonality** is the set of components and their interactions that are present in all configurations of the optimization system, which include the scheduler updating the schedule, the route time module answering requests from the schedule, and the dispatcher sending routing orders to the vehicles.
- The **variability** includes how the list of waiting shipments is prioritized, how the system calculates the cost of assigning each vehicle and driver combination to the pickups, how late pickups and dropoffs are handled, and how the system handles response time degradation.

By applying the SCV analysis to CONST we designed a PLA that enables the customization of its optimization engine for various domains.

The remainder of this paper is organized as follows: Section 2 describes the design of the Generic Eclipse Modeling System (GEMS) and how it addresses the challenges of creating MDD-based customization tools for PLAs such as CONST; Section 3 describes the Model Extension Language Editing Environment (MELEE) and uses a case study to show how it enables the creation of layers of abstraction between models; and Section 4 presents concluding remarks.

2 The Generic Eclipse Modeling System (GEMS)

To create PLAs rapidly, tools are needed to assemble and customize an organization's component assets. The Generic Eclipse Modeling System (GEMS) created by the Distributed Object Computing (DOC) Group at the Institute for Software Integrated Systems (ISIS) at Vanderbilt University is an Eclipse-based Meta-configurable Modeling Environment that enables developers to generate Domain-Specific Modeling Language (DSML) tools for customizing PLAs via the following capabilities:

- A visual interface that supports the creation of domain-specific modeling languages (DSMLs), i.e., GEMS contains a metamodeling environment that supports the definition of paradigms, which are type systems that describe the roles and relationships in particular domains.
- The creation of models that are instances of DSML paradigms within the same environment.
- Customization of such environments so that the elements of the modeling language represent the ele-

ments of the domain in a much more intuitive manner than is possible via third-generation programming languages.

- A flexible type system that allows inheritance and instantiation of elements of modeling languages.
- An integrated constraint definition and enforcement module that can be used to define rules that must be adhered to by elements of the models built using a particular DSML.
- Facilities to plug-in analysis and synthesis tools that operate on the models.

Each of these capabilities is shown in Figure 2 and discussed below.

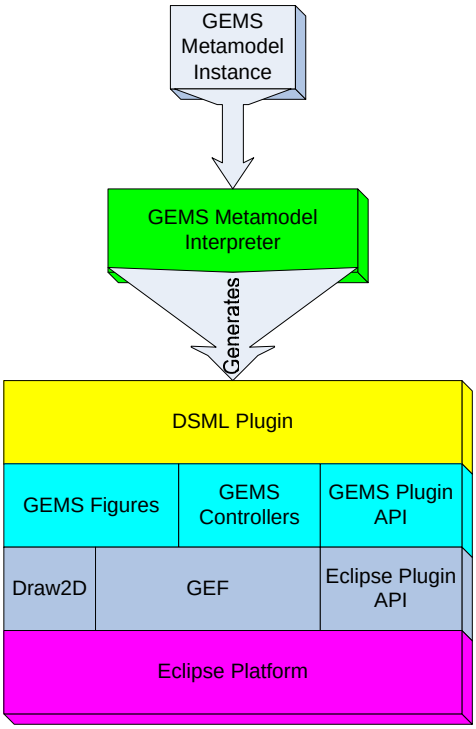

Figure 2. The GEMS Architecture

A key challenge to developing PLA customization tools is the formal specification of the PLA's DSML. Engineers must create a language to capture both the static and dynamic semantics. The GEMS metamodeling language allows developers to specify the common types in their system, the variability in instances of the types, the allowed containment compositions of the types, and the allowed connections between instances of types. These capabilities allow developers to apply SCV analysis to their PLAs and capture the results in a metamodel. This metamodel can then be interpreted by the GEMS tool to generate DSML(s) for customizing the PLA. By allowing developers to formally define the commonality/variability in the system and automatically generate a graphical customization tool that operates on that domain, GEMS significantly reduces the development effort require to develop a PLA by (1) generating a complete implementation of a tool for customizing the PLA, (2) ensuring that only legal PLA variants can be constructed in the tool, and (3) providing a modeling language intuitive to domain experts that customize the PLA.

In the remainder of this section, we discuss the design and functionality of the GEMS metamodeling environment. Each portion of the metamodeling system is examined in the context of the challenges it solves for building DSML customization tools for PLAs.

Section 2.1 Challenge 1: Allow DSML Tool Developers to Leverage OO Concepts to Construct Flexible DSMLs

Context. Extensive work has been done identifying how to decompose problems into object oriented systems. A good metamodeling environment must allow DSML developers to leverage the extensive knowledge of OO program design. By allowing developers to leverage OO design and best practices in their metamodels, existing patterns and development expertise can be reused.

Problem → The metamodeling language must support OO concepts, such as inheritance and polymorphism, and provide a mapping of these concepts to the meta-configured modeling environment.

In order to support OO designs and patterns, the metamodeling language must support inheritance and polymorphism. These concepts from the metamodeling language must in turn be mapped to the underlying meta-configurable environment. The meta-configurable environment must be able to evaluate constraint, visualization, and composition rules on both base types and their descendants.

Solution → Provide a metamodeling type system that leverages the polymorphism and inheritance facilities of the underlying OO programming language implementation.

To facilitate model reuse and maintenance, GEMS supports model types, instances, and type inheritance, similar to OO language concepts. GEMS provides two key types that can be defined: *atoms* and *models*. Atom types provide indivisible model units and may not contain other elements. Model types allow developers to specify entities that can contain other elements in the target DSML.

Model and atom types also support properties, which are data elements that compose the type. These are the same as the member variables of an OO class. Properties themselves have a simple type specified to constrain their value. GEMS supports `Enumeration`, `String`, `Integer`, `Decimal`, and `Boolean` types. To provide

more fine grained type constraints, GEMS also provides a built-in facility for extending these basic types using regular expression matching,. Both models and atoms support inheritance, which allows developers to provide polymorphic capabilities in the target DSML and facilitates the reuse of entities and provides for better abstraction. Inheritance applies to the properties, visualization rules, containment rules, and constraints of a DSML.

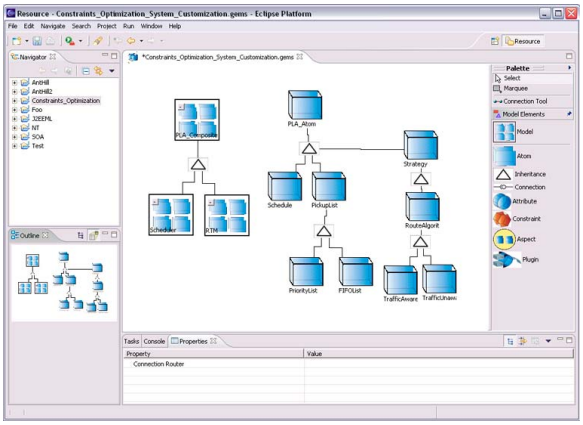

Figure 3. Metamodel for CONST Customization Tool

To create a customization tool for our CONST constraint optimization system, we first defined types for each of its configurable components identified during the commonality phase of SCV analysis. Figure 3 illustrates the metamodel for the CONST PLA customization tool. This figure shows how we used GEMS to model common components, including the route time module, scheduler, schedule, and route time calculation algorithms. The route time module can be configured to use either a route time algorithm that is aware of traffic delays or a faster calculation algorithm that ignores traffic delays. The traffic-aware algorithm is suited to contexts, such as assigning limousines to reservations, where lateness is measured in minutes. In a situation where shipments only need to arrive by multi-hour window within a day, such as scheduling trucks to freight shipments, the traffic-unaware algorithm may be more appropriate.

Section 2.2 Challenge 2: Ensure that Models are Correct by Construction

Context. A key benefit of using MDD tools to customize PLA variants is that the models are guaranteed to be correct by construction. Without guaranteeing that models are correctly constructed, developers cannot be certain that a valid PLA variant is being constructed. Correctly constructed models are even more crucial when multiple PLA customizers are working on the same variant. Without a properly constrained modeling tool, one PLA customizer can introduce subtle errors which ripple down the PLA customization chain. It is therefore crucial that the metamodeling tool used to construct the MDD PLA customization tools provide a mechanism for specifying constraints that should be applied to instances of the DSML.

Problem → The metamodeling language must provide an expressive constraint specification system that can satisfy a wide range of requirements.

Guaranteeing correctly constructed models requires runtime checking of both containment and connection constraints in the DSML tool. Often, the tool will also require checking complex context dependent constraints, such as the RTM may only be deployed with a traffic aware algorithm if it is connected to a traffic status data source. Providing a constraints language and evaluator that is both natural to the DSML tool developer and capable of covering a wide range of requirements is challenging.

Solution → Provide graphical specification of context-free constraints and a constraint specification language for context dependent constraints.

A crucial aspect of constructing variants in PLAs is ensuring the validity of the variants. PLAs present multiple ways in which a variant may be constructed incorrectly. For example, a variant may contain a *composition error*, which may be caused by composing elements that cannot be grouped together or by failing to compose elements that must be composed together. Another source of problems in PLA variants is the definition of interactions between PLA components, e.g., an interaction may be created in the variant between two components with incompatible interfaces. There are also complex validations that cross each of these areas, e.g., a component may only be able to interact with another component if it has a particular property set to a specific value.

The GEMS metamodeling language provides multiple techniques for defining constraints to ensure that only valid variants can be constructed within the generated DSML. Its constraint checking system takes an exclusionary approach, where only connections and containment relationships that are explicitly defined in the metamodel are allowed. The first constraint type is *cardinality*, which can be constructed for (1) *containment* by setting values on the containment association connection and (1) *connections* by setting the cardinality values on the connection declaration atom. At runtime, containment and connection constraints are evaluated before new connections and children are added to the model. Only valid connections and parent/child relationships are allowed.

In the metamodel for our CONST customization tool, we define a containment constraint that ensures exactly one scheduler component exists in any variant and that the scheduler component interacts with a single pickup list. We also define a containment constraint for the route time module to ensure it has exactly one route time cal-

culation algorithm that it uses. An attempt to create a system with no route time calculation algorithm or a scheduler that interacted with more than one pickup list will thus be flagged as an invalid variant.

To further ensure proper containment relationships and connection relationships in the generated DSML, GEMS supports the definition of type-based constraints. When a connection or containment relationship is declared between two entities within the metamodel, the relationship is constrained by the declared types of each entity. Only entities of that type, or of a more specialized type created through inheritance, can participate in the connection or containment relationship. These type-based constraints allow developers to ensure that only valid types are composed together and that all connections are between valid types. In the CONST DSML, for example, a route time module may only contain route time calculation algorithms. We define this constraint in the metamodel through a containment connection that ensures the generated DSML will uphold this constraint.

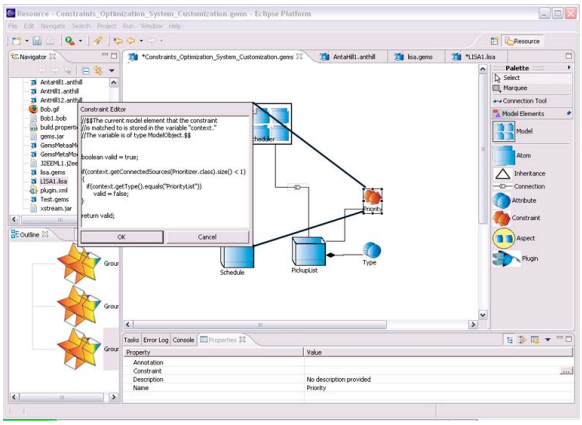

Figure 4. Complex Constraint for a PickupList

Some constraints that must be upheld do not depend solely on a single connection or containment relationship. For these complex constraints, GEMS provides a constraint entity that can be associated with a type. The constraint is evaluated anytime an action is performed on an instance of that type, such as adding a child or connection. To create the constraint, GEMS provides a constraint editor that evaluates any valid Java code that utilizes the GEMS model API. This constraint code is passed the model entity instance to which the constraint is being applied. The constraint code must use this instance to evaluate the validity of the entity's state and return a boolean value to indicate whether the constraint is honored or violated. Figure 4 shows a complex constraint defined for PickupList to ensure that a system with a prioritized pickup list is only created if there exists a component to determine the prioritization of the list.

Section 2.3 Challenge 3: Provide PLA Customizers with Intuitive Visualizations of Their Domain

Context. An important aspect of MDD tools is that they allow domain experts to manipulate entities familiar to their domain. In order for an entity to appear familiar to a domain expert, it must use the standard visual representation from its domain. Without the ability to display entities using the visual standards of their domain, domain experts are forced to operate in a less intuitive environment.

Problem → The metamodeling language must provide MDD tool developers with mechanisms for customizing the DSML's appearance and creating different views of the DSML.

To enable MDD tool developers to create useful DSML tools, the metamodeling tool must support the specification of the visual properties of the DSML's entities. The specification method must be flexible enough to cover a large number of domains but at the same time should only require tool developers to hand write code for very unusual visualizations. Striking this careful balance between flexibility and usability is challenging.

Solution → Provide intuitive default visualizations, graphical specification of common visual customizations, and mechanisms for completely overriding the default rendering mechanisms for unusual requirements.

By default, GEMS generates stock icons to represent the models and atoms in the generated DSML. Most DSML developers, however, will want to customize the rendering of the model entities so their look is specific to the domain. GEMS provides developers with the ability to customize the look of each type by providing icons for use in place of stock icons. Developers can also provide custom dynamic rendering of model entities. To facilitate dynamic rendering, GEMS provides an API developer can use to provide custom rendering classes for their visualization needs. Connections may also have their appearance customized. Both the endpoint styles (e.g., diamond, arrow, etc.) and the line style itself (e.g., dotted, color, etc.) can be customized through settings in the metamodeling language. The appearance of attributes can also be customized. Connection attributes can be configured to appear only in the attribute window or to be visible on the rendered connection.

A PLA can often have hundreds of customizable components. In these cases, visualizing the entire model in its entirety becomes too complex. In other cases, the model may have different possible views that are independent of each other. In these cases, a mechanism is needed to define different views on the model. Each view should

only expose the tools and entities relevant to it. Views can be leveraged to separate the roles of the domain experts by defining different views for each domain expert's role. Views also allow domain experts to focus on a specific aspect of the model, such as the physical or logical views.

The GEMS metamodeling language provides a facility for defining different views on the model. Each view in the DSML is modeled as an *Aspect*, which can be dragged into the model and have types associated with it via containment declarations. In the generated DSML editor, these views can be accessed to provide visual filters. Only types explicitly associated with the aspect through a containment connection in the metamodel will be visible on the tool palette and model canvas. Providing an aspect element allows developers to create multi-view editing environments rapidly and add new views easily as requirements change.

Figure 5 illustrates the creation of one view to customize the route time module and another view to customize only the scheduler. The RTM_View will create a model view in the generated DSML that allows the developer to view only the model entities related to the customization of the RTM. Similarly, the Scheduler's view will allow the developer to isolate and view only the entities for customization of the Scheduler, such as the PickupList and Schedule types.

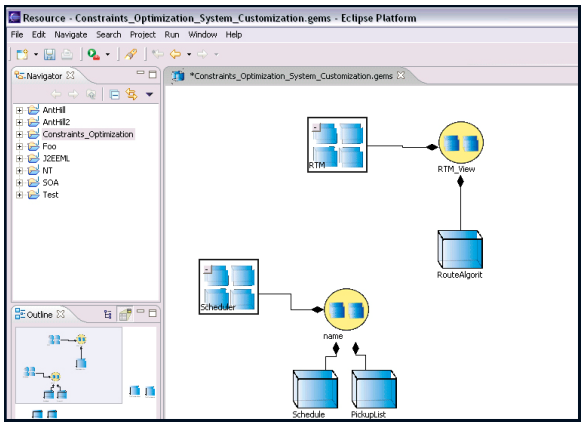

Figure 5. GEMS Aspects

Section 2.4 Challenge 4: Provide PLA Customizers with a Mechanism for Mapping DSML Instances to Physical PLA Artifacts

Context. An important aspect of MDD tools is that they allow domain experts to manipulate entities familiar to their domain. In order for an entity to appear familiar to a domain expert, it must use the standard visual representation from its domain. Without the ability to display entities using the visual standards of their domain, domain experts are forced to operate in a less intuitive environment.

Problem → A flexible model traversal mechanism must be provided that allows MDD tool developers to create model interpreters to generate source artifacts from DSML instances.

A key ability of MDD tools is the ability to generate source artifacts and perform model analyses. To facilitate this, the metamodeling environment must allow developers to create model interpreters that can be packaged with the generated DSML tools. This requires exposing a model traversal API to allow developers to analyze the model entities and map them to source artifacts. The domain experts must also be provided with a mechanism for accessing the registered model interpreters for their model type.

Solution → Provide a mechanism for programming API that allows MDD tool developers to access the Java objects implementing the DSML entities.

Creating variants for PLAs that must be handcrafted is tedious and error-prone. Often, the requirements and design of the variant are well understood, but incorrect interpretation of the specification by software developers yields implementations that do not match the requirements. Moreover, even developing implementations that meet the specification still requires significant software development time, testing, and debugging, and manual implementation of variants leads to rediscovery of existing and well understood solutions. As the complexity of PLAs increases and the time-to-market windows shrink, tools are needed to reduce the complexity of implementing a variant specification.

GEMS provides an interpreter framework developers can use to create code generators that operate on model instances of the generated DSML. By creating interpreters, developers can capture well-understood commonality and variability in the PLA in a reusable form. Implementations of the variant can be produced automatically by interpreting model instances, which allows variants to be generated quickly as requirements change or as new variants are defined. Code generation also saves significant software development, validation, and debugging time since after a correct set of interpreters has been constructed and validated, all generated implementations of variants will be correct-by-construction.

The GEMS interpreter framework provides a Java API that allows interpreter developers to traverse instances of the model entities. This API includes methods for querying the children of each model, querying attribute values, and querying connections. The GEMS Metamodel interpreter generates the Eclipse plug-in artifacts required to implement the modeled PLA customization tool. The generated code includes the classes needed for the Draw2D and GEF models, controllers, and views. It also includes the XML plug-in descriptor required to inte-

grate the customization tool into Eclipse. For customizing our CONST PLA customization tool, the interpreter generated approximately 31 classes, 500 lines of Java code, and 70 lines of XML, all of which would have been created manually by developers using a traditional manual development process. For our CONST customization tool, we developed multiple interpreters, leveraging our previous work on the Enterprise Jave Bean modeling tool, J2EEML [7], to generate artifacts to automate variant implementation. These interpreters include numerous artifacts, including XML descriptor and configuration files, Enterprise Java Beans, ANT deployment scripts, Java Docs, testing frameworks, and JNDI lookup code.

2.5 Applying GEMS to CONST

We developed the CONST case study to illustrate the advantages of using the GEMS-based PLA customization tool to create constraint optimization system variants. A meta-model describing the problem domain was created in GEMS. The GEMS plugin-generator was then used to create an Eclipse plugin for creating CONST variants. We then performed experiments comparing the reduction in development effort provided by the GEMS-based CONST customization tool versus traditional development methods.

The initial generated implementation of this case study created by the CONST customization tool contained several thousand lines of Java code. The generated EJB implementations accounted for nearly 75% of the complete code base, the test framework accounted for 20%, and the adaptive glue code accounted for 5%. Using a traditional development process, much of this code would have been developed manually. With our PLA customization tool, in contrast, all code except for the business logic and testing logic was generated initially from our specification, which accounted for approximately one-third of the code required to implement the Java classes for the application.

Using our highway freight scheduling case study, we evaluated the difficulty of creating several new variants. The first variant was constructed by customizing the PLA to add monitoring and adaptation logic. We refactored the design to monitor the response times of the route time module component. Adjusting the design using the PLA customization tool and regenerating the implementation took approximately five mouse clicks and resulted in the generation of ~20 new lines of source code that correctly mirrored the specification and was correct-by-construction. Without GEMS's code generation tools, in contrast, each change that required the addition of one new connection in the model would have required a manual change to the implementation and more testing to attain the same level of confidence. In a large-scale development effort with many changes, using the PLA customization tool would therefore reduce refactoring effort significantly.

To evaluate the impact of design refactoring on the route time model monitoring variant, we modified its initial design by changing its response time analysis into a hierarchy of average and maximum response time analyses. We also added adaptive logic to adapt to poor response times. The adaptive logic took the following series of actions: the first action was to suspend the processing of incoming requests, the second action taken was to change the algorithm used by the route time module to one that is less precise but faster, and the final action taken by the adaptation was to resume processing requests. The refactoring in the PLA customization tool was straightforward and took ~12 mouse clicks. The change generated ~75 new lines of code, which minimized the complexity of the design change and implementation update. Again, for large development projects without MDD tool support, many such changes would occur and hence the manual redevelopment effort would be much higher.

3 Creating Different Abstractions for Each Domain Expert

PLA variants are constructed by having a group of domain experts customize different portions of the product. In an DSML-based process, each domain expert needs a modeling tool that focuses on their specific domain. The domain experts' modeling tools must, however, remain interoperable, e.g., so that domain experts can pass their models on to the next domain expert in the customization chain. Each domain expert may customize larger and larger pieces of the system and may therefore not need to see all the low-level details configured by the previous domain experts.

PLA variants can be constructed through a variety of means. In some scenarios, one domain expert may need to customize the work of another domain expert. In our CONST PLA, for instance, one domain expert creates the EJBs implementing the system components and models their library dependencies. Another domain expert is then responsible for determining which of these EJB components are deployed to compose different variants. The EJB domain expert need not know about the intricacies of deploying a variant. The deployment expert should not be required to understand all the artifact dependencies of each component. Instead, the deployment expert's customization tool should ensure that each EJB is deployed with its required artifacts without exposing this information to the deployment expert. Further complicating the problem is that the tools of the domain experts must be interoperable. Ideally, both (all) tools operate on the same underlying model.

This section examines R&D challenges associated with creating layers of abstraction on top of models. For each challenge we will (1) explain the context in which the

challenge arises, (2) identify the specific problems that must be addressed, and (3) outline an approach that helps resolve the challenge and show how we could apply these solutions in our CONST PLA.

To present the challenges associated with creating layers of abstraction, we use a case study based on CONST. CONST presents significant challenges for the deployment and configuration of its distributed components, including the RTM and Scheduler It is therefore important that a PLA customization tool be provided, which can guarantee that a variant of the optimization system is correctly deployed and configured. We extended an existing deployment and configuration tool, the FireAnt model driven deployment and configuration system to create a deployment and configuration tool for CONST. We use the extension of FireAnt as the case study for illustrating our model abstraction technologies.

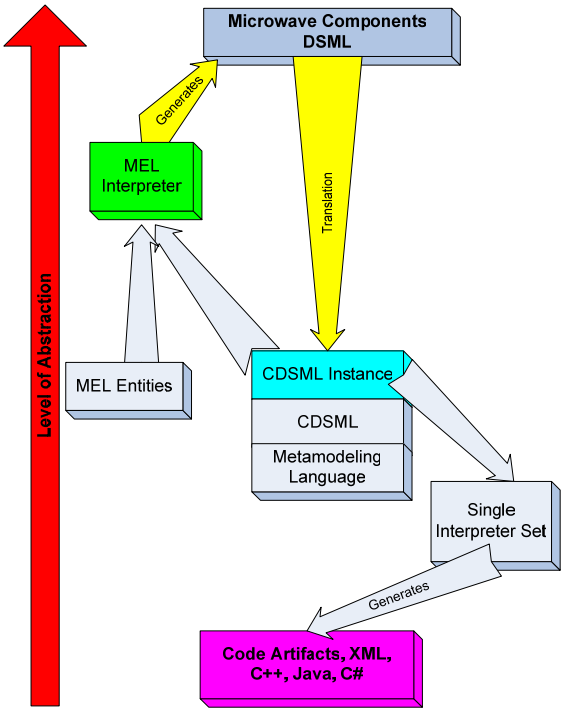

Figure 6. MEXTL Abstraction Architecture

Section 3.1 The Model EXTension Language (MEXTL)

The *Model EXTension Language* (MEXTL) is a DSML that allows developers to annotate an existing instance of a DSML, which we call the *Basis Underlying Model* (BUM), to provide a layer of abstraction on top of it. MEXTL provides mechanisms for specifying which portions of the *Basis Underlying Modeling Paradigm* (BUMP) to hide in the new DSML. It also contains facilities for describing points of variability in the new DSML and how they are mapped to the BUMP. Finally, MEXTL allows developers to mark which groups of elements from the BUM should serve as atomic units in the new DSML. Figure 6 illustrates the process by which an existing DSML instance is extended using MEXTL to create a new modeling paradigm.

We have developed extensions to GEMS that provide a graphical editing environment for MEXTL. The environment supports the importation of existing instances of models, developed with GEMS, and their annotation with MEXTL entities. The editor provides mechanisms for viewing the underlying model instance and MEXTL elements separately. Interpreters for models annotated with MEXTL entities can generate new DSMLs, GEMS-based Eclipse Plug-ins, and translation interpreters from the new DSML to the BUMP.

Section 3.2 Challenge 1: Expose the Same Model Elements Through Different Abstractions to Each Domain Expert

Context. PLA Variants are constructed by having a group of domain experts that customize different portions of the product. Each domain expert needs a modeling tool that focuses on their specific domain. The domain experts' modeling tools must, however, remain interoperable. Domain experts must be able to pass their models on to the next domain expert in the customization chain. Often, each domain expert will customize larger and larger pieces of the system and will not need to see all the low level details configured by the previous domain experts.

Problem → Elements of a model must be exposed to two domain experts through different abstractions. PLA variants can be constructed through a variety of means. In some scenarios, one domain expert may need to customize the work of another domain expert. In our CONST PLA, one domain expert creates the EJBs implementing the system components and models their library dependencies. Another domain expert is then responsible for determining which of these EJB components are deployed to compose different variants. The EJB domain expert does not need to know about the intricacies of deploying a variant. The deployment expert should not be required to understand all the artifact dependencies of each component. Instead, a mechanism should exist to allow the deployment expert's customization tool to ensure that each EJB is deployed with its required artifacts without exposing this information to the deployment expert. Further complicating the problem is that the tools of the domain experts must be interoperable. Preferably, both tools operate on the same underlying model.

Solution → Forward an entity of one model to another model but with a different name and controlled access to its attributes. The Model EXTension language

(MEXTL) provides entities that can be added to instances of existing metamodels to allow them to become the metamodel for a new DSML. MEXTL is the language that specifies what entities in the basis model should be exposed in the new DSML. It also specifies how each entity should be exposed in the new DSML. In some cases, an entity may need to be exposed in a manner that hides its constituent parts. In other cases, the entity may allow controlled access to its constituents.

MEXTL provides the means to directly pass entities from the BUMP to the new DSML. Entities in the BUMP can be connected to a *Forwarding Atom* that indicates that the entity and its functionality should be copied into the new DSML. In the new DSML, creation of this entity will result in the creation of an entity with the same type as the forwarded entity from the BUMP. In the new DSML, however, the entity will automatically have its properties initialized to the same values as those of the instance that the forward was based on. The forwarded entity may also have its name and appearance changed in the new DSML. For each property on the forwarded entity, the forwarding atom allows the developer to specify if the property is read-only, read-write, or hidden. This allows developers to expose the entity in a controlled manner. The declared containment relationships and connection relationships for the forwarded entity are maintained in the new DSML if at least one entity with a type, compatible with the other end of the relationship, is forwarded.

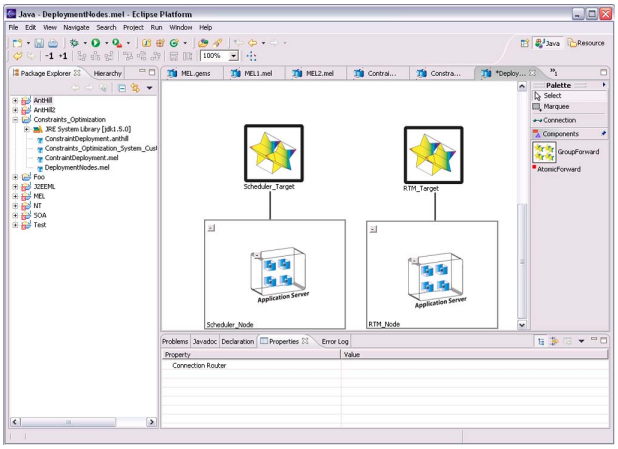

Figure 7. Atomic Forwarding of the Deployment Nodes for the Scheduler and RTM

For the constraints optimization system deployment and configuration tool, the requirements for the application infrastructure, such as requiring an EJB application server, do not change substantially. Thus we used atomic forwarding to expose the target nodes for the constraint optimization system in the configuration and deployment tool. In the configuration and deployment tool, developers only specify the physical address of the node. The MEXTL extension hides the other details that the deployer does not need to be concerned with. Figure 7 shows the MEXTL mode where the deployment nodes are being atomically forwarded.

Section 3.3 Challenge 2: Expose a Group of Entities Separately to One Domain Expert and as an Atomic Unit to Another Domain Expert

Context. Often, one PLA customizer may compose low level entities into components that are the basic entities over which another PLA customizer operates. In the CONST deployment tool, the EJB customizer specifies which EJBs and libraries form which components. The deployer then maps these components to physical nodes. The deployer, however, needs to only know about the components and not their low level library dependencies and EJB compositions. The EJB customizer must pass these compositions of EJBs and libraries on to the deployer in a manner that preserves this dependency information but hides it from the deployer.

Problem → Allow one domain expert to create compositions of entities and another domain expert to configure the compositions without seeing their constituents. If each entity from the composition is forwarded separately to the new DSML, the next customizer in the chain will be exposed to unnecessary low-level details. To provide this type of abstraction, the modeling tool must provide a means for a group of entities to masquerade as an atomic unit in higher levels of abstraction.

Solution → Forward entities of one model to another model as an atomic unit. To allow composite components to be forwarded to the new DSML, MEXTL provides the *Composite Forwarding Atom*. When an entity is connected to the composite forwarding atom, it and all of its connected and contained components are exposed as a new element in the generated DSML. As with the atomic forwarding, the newly forwarded group can be given a new name and appearance. The composite element allows the developer to specify whether the forwarded entity in the new DSML is exposed as an atom or model. If it is exposed atomically, the entities connected and contained components will not be accessible. If the entity is exposed as a model, each of the constituent components will be accessible.

To hide the jar and ear dependencies of the route time model and scheduler, we used composite forwarding to expose the route time module and its artifacts as a single atomic entity in the new DSML. The scheduler was also forwarded in this manner. This allowed component deployers to focus on where the application pieces would reside and not worry about whether or not the applica-

tion pieces themselves were correctly constructed. The application developers constructed the dependency information for the application pieces. Figure 8 shows the MEXTL model where the route time module and scheduler composites are being forwarded.

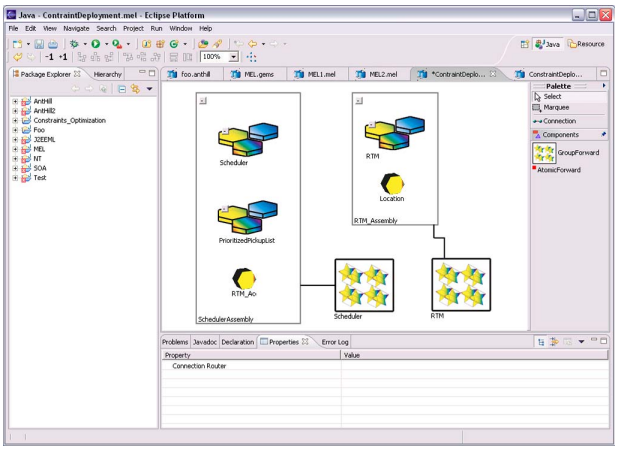

Figure 8. Group Forwarding of the Scheduler Assembly and RTM Assembly

Section 3.4 Challenge 3: Forwarded Compositions and Entities May Have New Connection and Other Constraint Requirements

Context. One PLA customizer may need to declare constraints on the compositions they create. In the CONST deployment tool, the EJB component customizer must be able to specify which components can be co-located and which components cannot. The EJB customizer must also specify which components must be deployed for each system and their composition rules. Without these rules, the deployment expert will be able to compose invalid CONST systems.

Problem → Mechanisms must exist to declare new constraints for forwarded entities. Each customizer must have a mechanism to pass constraints on to the next customizer in the chain. Preferably, these constraints should be enforced by the tool chain to ensure that they are applied. Constraints, however, are usually specified only in the metamodel and apply to general types. Therefore, a mechanism must be created that allows constraints to be applied to specific entity instances within a model.

Solution → Allow developers to declare connections between forwarded entities using the metamodeling language. After forwarding components into the new DSML, developers may still need to declare new entities that will be available. MEXTL allows developers to use the GEMS metamodeling language to specify new connections between the forwarding atoms that will apply to the forwarded entities in the new DSML. MEXTL also provides developers with the ability to declare new model and atom types that inherit from the forwarded entities. Inheriting from forwarded entities allows developers to specify properties and relationships that only make sense in the new DSML.

The GEMS metamodeling visualization capabilities are fully supported in MEXTL. Developers may create aspects or views on the forwarded entities that appear in the new DSML. The creation of aspects works as described in Section 2.3.

4 Related Work

The *Generic Modeling Environment* (GME) [9] [10] is a MDD environment that supports the rapid development of DSMLs. GEMS provides several capabilities not found in GME. GME does not provide a model extension language that allows layers of abstraction to be built on top of existing model instances. GME's approach to providing separate environments to each domain expert is its concepts of paradigm sheet and references. References allow one part of the model to refer to another portion of the model. They do not, however, allow for the creation of new constraints on the referenced elements as in MEXTL. Paradigm sheets allow multiple paradigms to be shared in the same modeling environment. Paradigm sheets do not enforce the separation of concerns between models. Each domain expert can view and edit the paradigm sheets of the other domain experts. MEXTL allows for an enforced and clean separation of modeling concerns.

The Eclipse Foundation's *Graphical Modeling Framework* (GMF) [11], is an emerging MDD tool development framework designed to facilitate the creation of DSMLs. Unlike GEMS, however, GMF does not provide a graphical editor for constructing metamodels. Instead, it takes metamodels created via a separate tools and generates editing environments for the metamodel. GMF is also early in development and does not yet have a concrete implementation. GMF does intend to support allowing multiple model entities to masquerade as a single entity but does not provide a graphical metamodeling environment to specify these compositions and to further extend them as MEXTL does.

Microsoft's Domain-Specific Language Tools [13] is another toolsuite for developing MDD tools that supports the creation of DSMLs. As with GMF, the Corona tools do not provide a graphical means to define metamodels. Moreover, Corona does not support a model extension language like MEXTL.

5 Concluding Remarks

In theory, PLAs can reduce the effort associated with developing a group of component applications. In practice, it is hard to develop the PLA customization tools required to reduce effort. Without PLA customization tools, developers must reason about complex sets of QoS requirements and large configuration spaces, which

makes customization tedious and error-prone. With PLA customization tools, developers can more effectively reason about the correctness of a PLA variant and ensure that key QoS requirements are met.

Developing PLA customization tools themselves is hard today, however, due to limited time-to-market and complex graphical modeling requrirements,. It is therefore essential that MDD tools be created that allow for the rapid construction of DSML tools for customizing PLAs. The GEMS MDD tool described in this paper enables developers to graphically describe their target domain and ensures that PLA customization tools generate correct models the domain. GEMS allows developers to focus on the design of their target DSML, rather than on low-level details of integrated development environment APIs and drawing packages. By allowing developers to focus on a higher level of abstraction and automating the implementation of PLA customization tools, GEMS reduces development and evolution effort. Moreover, GEMS capability to regenerate implementations of the DSML tool as requirements change also significantly reduces development effort.

Well-crafted PLA customization tools enable each domain expert to focus on their portion of a PLA. Insulating domain experts from complexities they are not concerned with, improves the usability of the tool, reduces the learning curve, and helps clarify developer roles. MEXTL provides an automated method for building layers of abstraction on models. Developers can use MEXTL to create PLA customization tools rapidly that separate out each domain specialty and shield domain experts from areas that are not their concern.

In future work, we are developing advanced declarative programming, knowledge base, and querying functionality for GEMS, as well as addressing the challenges of maintaining model assets when the underlying metamodel changes. The GEMS software and examples are available in open-source format from www.sf.net/projects/gems.

References

[1] P. Clements, L. Northrop, *Software Product Lines, Practices, and Patterns*, Addison-Wesley, Boston, 2002

[2] D. Oppenheimer, A. Ganapathi, D. Patterson, "Why do Internet services fail, and what can be done about it?", "USENIX Symposium on Internet Technologies and Systems,USENIX Symposium on Internet Technologies and Systems," March 2003.

[3] N. Wang, D. Schmidt, A. Gokhale, C. Rodrigues, B. Natarajan, J. Loyall, R. Schantz, and C. Gill, "QoS-enabled Middleware," in *Middleware for Communications*, edited by Q. Mahmoud, Wiley and Sons, New York, 2003.

[4] K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, and D. 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.

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

[6] J. Coplien, D. Hoffman, D. Weiss, "Commonality and Variability in Software Engineering," *IEE Software*, 15(6), November/December 1998

[7] J. White, D. Schmidt, A. Gokhale, "Simplifying the Development of Autonomic Enterprise Java Bean Applications via Model Driven Development.," In: Proc. ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (2005), Seattle, WA

[8] Apache Foundation: Apache Ant. http://ant.apache.org

[9] Ledeczi, A., Bakay, A., Maroti, M., Volgysei, P., Nordstrom, G., Sprinkle, J., Karsai, G, "Composing Domain-Specific Design Environments," IEEE Computer, Nov. (2001)

[10] A. Ledeczi, "The Generic Modeling Environment," In: Proc. Workshop on Intelligent Signal Processing (2001), Budapest, Hungary

[11] The Eclipse Foundation: The Graphical Modeling Framework. http://www.eclipse.org/gmf/

[12] G. Edwards, G. Deng, D. Schmidt, A. Gokhale,, B. Natarajan, "Model-driven Configuration and Deployment of Component Middleware Publisher/Subscriber Services," In: Proceedings of the 3rd ACM International Conference on Generative Programming and Component Engineering (2004), Vancouver, CA

[13] Microsoft Coroporation. Domain-Specific Language Tools, http://lab.msdn.microsoft.com/teamsystem/workshop/dsltools/default.aspx