TO BE FILLED IN BY THE EDITOR
Heading: Introduction to the Generic Eclipse Modeling System 2.0.5
Subheading: Using GEMS for Developing Graphical Modeling Tools for Eclipse
Standfirst: Graphical Model-Driven Engineering (MDE) tools have become extremely popular in the development of applications for a large number domains. In many cases, however, an organization does not have the resources or time available to develop a graphical modeling environment from scratch using the Eclipse Modeling Framework (EMF), Graphical Editor Framework (GEF), or Graphical Modeling Framework (GMF). In other situations, the complexity of the domain limits the feasibility of using a graphical model to describe a domain solution. The Generic Eclipse Modeling System (GEMS), which is part of the Eclipse Generative Modeling Technologies (GMT) project, helps developers rapidly create a graphical modeling tool from a visual language description or metamodel without any coding in third-generation languages. GEMS automatically generates the requisite EMF, GEF, and GMF code required to implement the editor from a metamodel. GEMS also provides extensive capabilities for expressing modeling guidance and performing optimization. Finally, graphical modeling tools created with GEMS automatically support complex capabilities, such as remote updating and querying, template creation, styling with Cascading Style Sheets (CSS), and model linking.
Author Name: Jules White
Author Bio: Jules White is a researcher in the Distributed Object Computing (DOC) group at Vanderbilt University's Institute for Software Integrated Systems (ISIS). Mr. White's research focuses on reducing the complexity of modeling complex domains. Before joining the DOC group, he worked in IBM's Boston Innovation Center. Mr. White is the head of development for the Generic Eclipse Modeling System (GEMS).
Author Name: Douglas C. Schmidt
Author Bio: Douglas C. Schmidt is a Full Professor in the Electrical Engineering and Computer Science (EECS) Department, Associate Chair of the Computer Science and Engineering program, and a Senior Research Scientist at the Institute for Software Integrated Systems (ISIS) at Vanderbilt University, Nashville, TN. For the past two decades, he has led pioneering research on patterns, optimization techniques, and empirical analyses of object-oriented and component-based frameworks and model-driven development tools that facilitate the development of distributed middleware and applications. Dr. Schmidt is an expert on distributed computing patterns and middleware frameworks and has published over 300 technical papers and 6 books that cover a range of topics including high-performance communication software systems, parallel processing for high-speed networking protocols, real-time distributed object computing, object-oriented patterns for concurrent and distributed systems, and model-driven development tools. In addition to his academic research, Dr. Schmidt has over fifteen years of experience leading the development of widely used, open-source middleware platforms (www.dre.vanderbilt.edu) that contain a rich set of components and domain-specific languages that implement key patterns for high-performance distributed systems. Dr. Schmidt received his Ph.D. in Computer Science from the University of California, Irvine in 1994. URL: www.dre.vanderbilt.edu/~schmidt. Email: schmidt@dre.vanderbilt.edu.
Author Name: Andrey Nechypurenko
Author Bio: Andrey Nechypurenko is a …... at Siemens Corporate Technology SE2 in Munich, Germany.
Author Name: Egon Wuchner
Author Bio: Egon Wuchner is a …... at Siemens Corporate Technology SE2 in Munich, Germany.
Character Count:
Figures: 13
Issue:
Publication Month:
Section:

Introduction to the Generic Eclipse Modeling System

by Jules White

Developing a Graphical Modeling Tool for Eclipse

Graphical Model-Driven Engineering (MDE) tools have become extremely popular in the development of applications for a large number domains. A wide range of frameworks support development of MDE tools for Eclipse. The Eclipse Modeling Framework (EMF) provides an object graph for representing models, as well as capabilities for (de)serializing models in a number of formats, checking constraints, and generating various types of tree editors for use in Eclipse. The Graphical Editor Framework (GEF) and Draw2D provide the foundations for building graphical views for EMF and other models types. Finally, the Graphical Model Framework (GMF) for Eclipse provides a rich framework for reducing the complexity of developing an GEF/EMF-based modeling tool. These frameworks are designed to provide the degrees of flexibility needed to support products, such as IBM's Rational Architect.

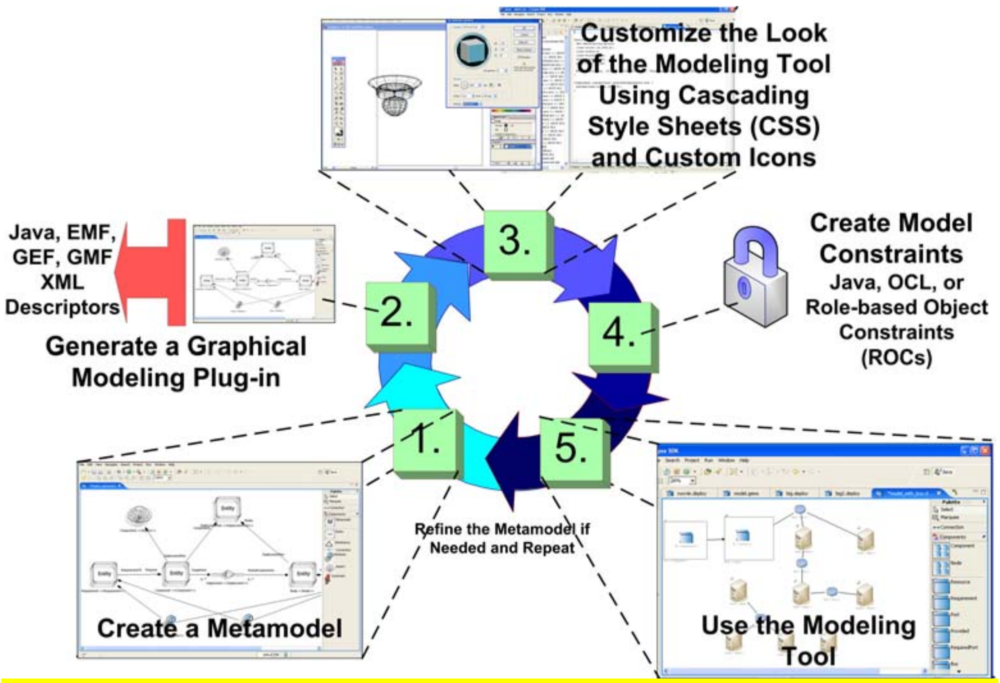
MDE tools are rarely simple to develop, however, and are generally reserved only for the highest payoff domains. Regardless of the framework used, developers must still produce a significant amount of XML and Java code to construct a working graphical modeling tool. Smaller organizations tend to lack the required development expertise to implement a graphical modeling tool using EMF, GEF, and GMF. Even in situations where an organization can develop a graphical modeling tool from scratch, the time spent performing these activities subtracts from the time available for developing the true assets of these tools; the complex domain validation, code-generation, optimization, and simulation capabilities. Moreover, there is a significant cost associated with maintaining a graphical modeling tool infrastructure, as understanding of the domain deepens and the tool and DSML requirements change.

The Generic Eclipse Modeling System (GEMS), a part of the Eclipse Generative Modeling Technologies (GMT) project, helps developers rapidly create a graphical modeling tool from a visual language description (metamodel) without any coding in third-generation languages. GEMS is an open-source project, based on the Eclipse License, that has been developed in conjunction with Siemens CT SE2, IBM, and Prismtech. GEMS gives developers the ability to graphically describe the domain-specific modeling language(s) (DSML) they wish to create a modeling tool for and automatically generate the requisite EMF, GEF, and GMF code required to implement an Eclipse-based plug-in for creating and editing instances of the DSML. GEMS pulls together these various Eclipse modeling frameworks and uses standardized naming conventions, file formats, and other simplifications to make coding an editor from scratch unnecessary. Even though GEMS uses multiple techniques to allow developers not to write code unless they want to, the decisions and default visualizations used by GEMS can still be overridden through the use of Cascading Style Sheets (CSS), extending GEMS extension-points and other facilities, such as model templates and remote updating mechanisms.

GEMS substantially reduces the cost of developing a graphical modeling tool by allowing developers to focus on the key aspects of their tool: the specification of the DSML and the intellectual assets built around the use of the language. The infrastructure to create, edit, and constrain instances of the language is generated automatically by GEMS from the language specification. As the language specification changes, GEMS can regenerate the Java, EMF, GEF, GMF, and XML descriptors required to edit the new language. Moreover, GEMS allows the separation of language development, coding, as well as "look and feel" development, such as changing how modeling elements appear based on domain analyses.

A GEMS-generated graphical modeling tool can have both static and dynamic visual behaviour specified through stylesheets. Graphic designers or developers can create styles and icons that should be applied to elements when they are in specific states. For example, a developer building a tool for specifying the deployment of software components to nodes could develop separate icons and styles for drawing a component when it is deployed and undeployed. The CSS capabilities of GEMS will be discussed in detail in the Section "Customizing the Look of the Modeling Tool with CSS."

The process for developing and using a graphical modeling tool using GEMS is based on the Model Integrated Computing paradigm [5] developed at Vanderbilt University and originally implemented in the Generic Modeling Environment (GME) [6]. This process can be seen in Figure 1.

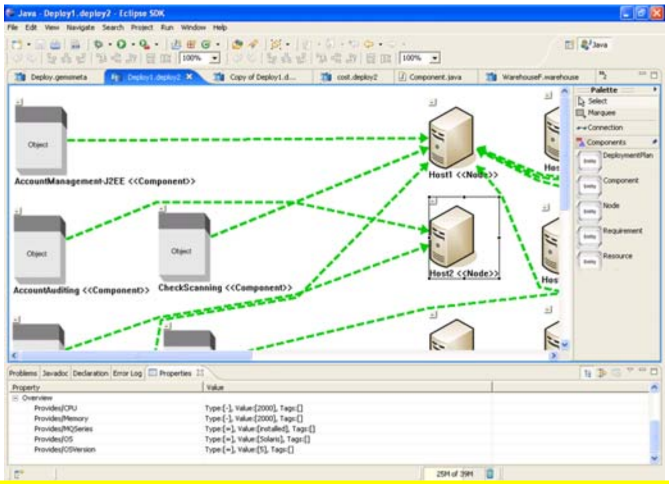

Customize the Look of the Modeling Tool Using Cascading Style Sheets (CSS) and Custom Icons
Java, EMF, GEF, GMF XML Descriptors
Generate a Graphical Modeling Plug-in
Create Model Constraints
Java, OCL, or Role-based Object Constraints (ROCs)
1.
2.
3.
4.
5.
Refine the Metamodel if Needed and Repeat
Create a Metamodel
Use the Modeling Tool

INSERT GEMS_Dev_Cycle_w.jpg
Figure 1: The GEMS Development Cycle

The first step for building a graphical modeling tool with GEMS is to define a metamodel for the DSML. This metamodel describes the graphical entities, connection types, attributes, and other visual syntax information needed by GEMS. In step two, the user invokes a code generator the produces the EMF, GEF, GMF, XML etc. needed to create a plug-in for editing the DSML described by the metamodel. Graphic designers or developers create custom icons and CSS styles for the DSML elements in the third step. For the fourth step, developers specify the constraints on the model that GEMS uses to ensure that only correct models are built with the model. The constraints generally involve domain information that requires a constraint language to express properly. Finally, domain experts use the modeling tool produced by GEMS to construct models of the domain. If the DSML is sufficiently expressive that no modifications are needed for the metamodel, model interpreters (or code generators) are developed to produce software artifacts (such as Java code or XML descriptors), run simulations, or execute the model. The following sections describe each of these steps in detail.

Creating a GEMS Metamodel

The core of a GEMS-based modeling tool is a metamodel describing the syntax of the DSML. As a case study, we describe the development of a tool, called AUTODeploy, for specifying the deployment of software components to nodes in a data center. AUTODeploy allows IT professionals to describe each software component they wish to deploy in their data center, the requirements of the node hosting each component, the nodes available in the data center, and the resources available on each node. AUTODeploy can also specify where components should be deployed by creating connections between components and nodes. Finally, this modeling tool can automatically deduce and create a valid deployment for all of the components. Figure 2 shows a screenshot for AUTODeploy that we will produce using GEMS.


Java - Deploy1.deploy2 - Eclipse SDK
AccountManagement-J2EE <<Component>>
AccountAuditing <<Component>>
CheckScanning <<Component>>
Host1 <<Node>>
Host2 <<Node>>
Palette
Select
Marquee
Connection
Components
DeploymentPlan
Component
Node
Requirement
Resource

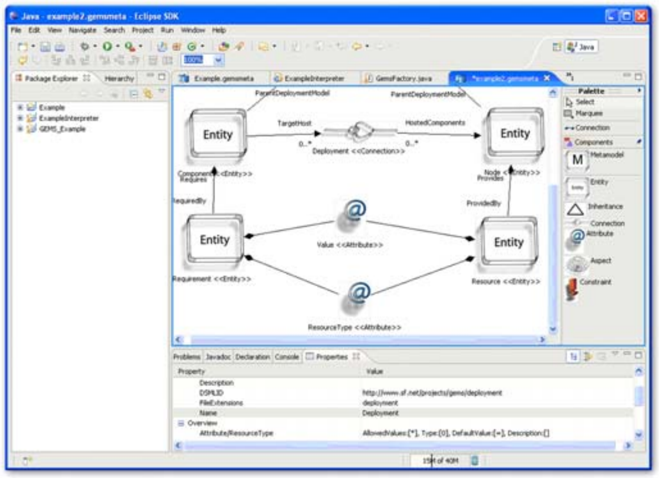
Property	Value
Overview	
Provides/CPU	Type:[-], Value:[2000], Tags:[]
Provides/Memory	Type:[-], Value:[2000], Tags:[]
Provides/MQSeries	Type:[=], Value:[installed], Tags:[]
Provides/OS	Type:[=], Value:[Solaris], Tags:[]
Provides/OSVersion	Type:[=], Value:[5], Tags:[]

INSERT autodeploy.eps

Figure 2: A GEMS-based Software Component Deployment Tool

Examining the finished AUTODeploy product we are creating helps clarify how the various parts of the GEMS development cycle fit together and map to AUTODeploy, which has five different types of model entities or elements. The first entity is the *DeploymentPlan*, which is represented as the main white canvas containing the *Nodes* and *Components*. The DeploymentPlan is the root entity in our model. Node and Component entities can be seen on the right and left hand sides of the model, respectively. Two more entity types are present but not visible in the screenshot. *Requirements* are child entities of Components that can be seen by expanding a Component by clicking on the button in the upper left hand corner of each Component. *Resources* are child entities of Nodes and are also not visible. We can, however, see evidence of the Resources in the properties pane of the tool.

The *Overview* attribute in the properties pane visible in the bottom of Figure 2 provides a brief outline of the children contained by each entity and the attributes of the children. CPU, GeoDB, and OS are all Resources contained by Host3. The finished AUTODeploy tool contains connections between Components and Nodes. These are Deployment connections that were defined in the metamodel. Figure 3 shows the complete metamodel for our software component deployment tool.

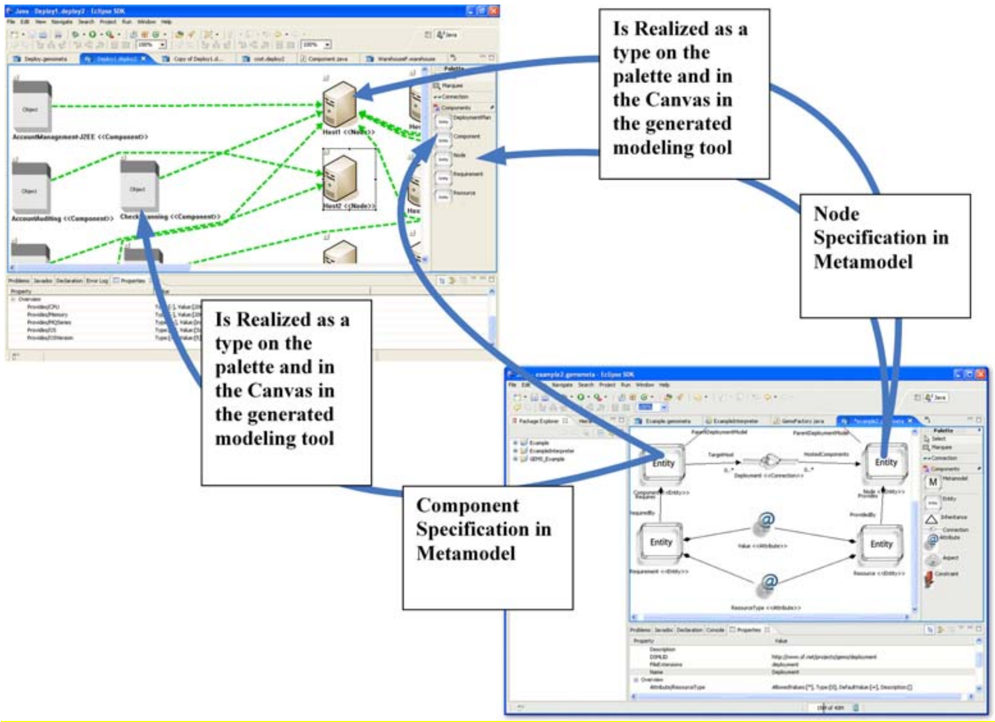


Property	Value
Description	
DSMLID	http://www.sf.net/projects/gems/deployment
FileExtensions	deployment
Name	Deployment
Overview	
Attribute/ResourceType	AllowedValues:[*], Type:[0], DefaultValue:[=], Description:[]

INSERT metamodel.eps

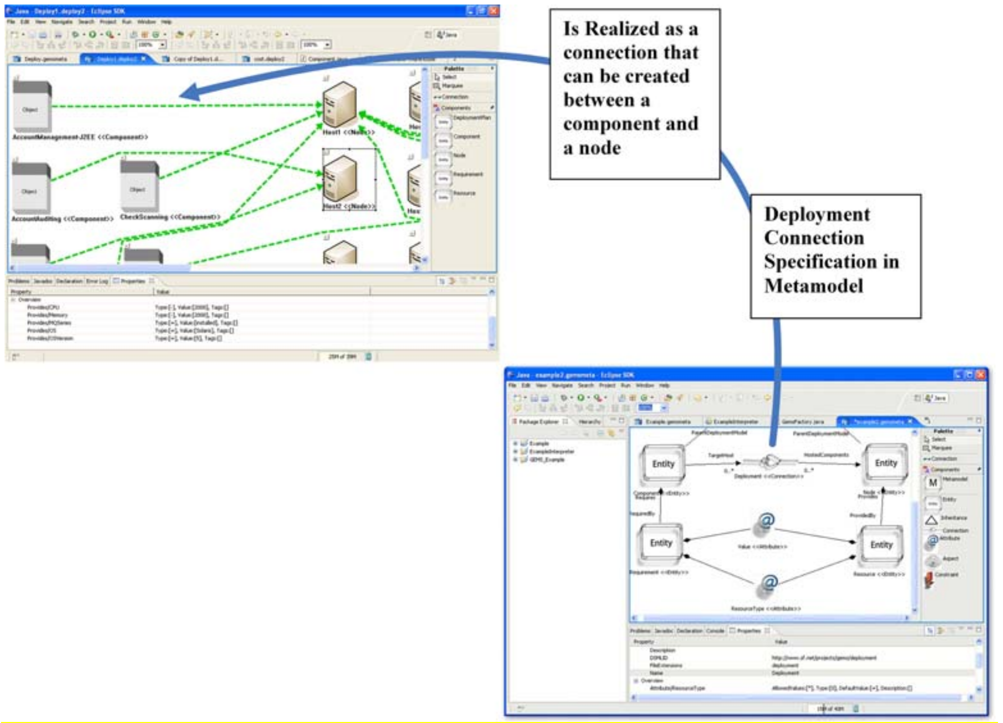
Figure 3, The Complete Software Component Deployment Metamodel

The *Entity* elements in the metamodel correspond to the entities we described in AUTODeploy. The metamodel can be viewed as a class diagram for the instances of the model entities seen in AUTO Deploy. The Entity elements in the figure above correspond to the types in our DSML. The *Atrributes*, visualized with the "@" sign, represent the properties of the types that can be set in the property pane. The *Connection* element, called Deployment, corresponds to the Deployment connection type in AUTODeploy. Figures 4, 5, 6, and 7 illustrate these mappings.

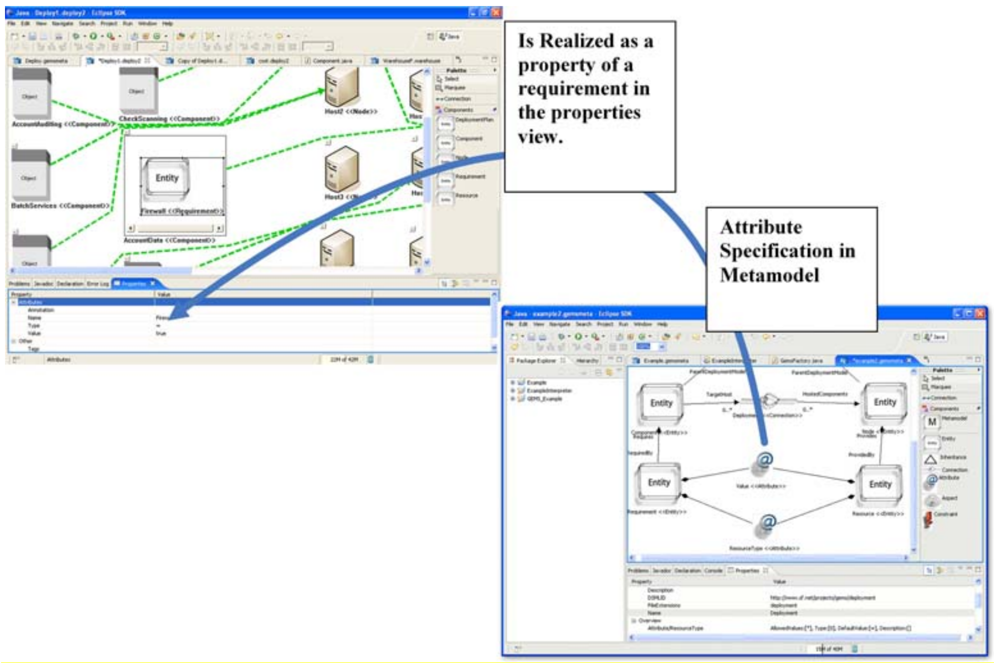

Is Realized as a type on the palette and in the Canvas in the generated modeling tool
Node Specification in Metamodel
Is Realized as a type on the palette and in the Canvas in the generated modeling tool
Component Specification in Metamodel
Entity
Entity
Entity
Entity

INSERT entity_mapping.jpg

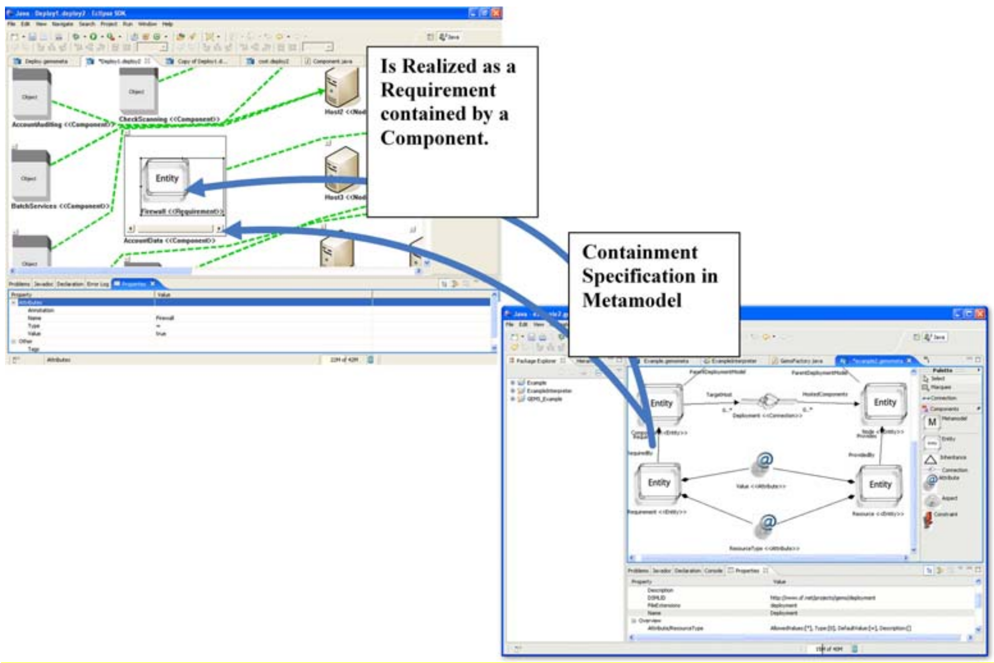
Figure 4: Realization of Entities in AUTODeploy


Is Realized as a connection that can be created between a component and a node
Deployment Connection Specification in Metamodel
Entity
Host1 <<Node>>
Host2 <<Node>>
AccountManagement-J2EE <<Component>>
AccountAuditing <<Component>>
CheckScanning <<Component>>
Deployment <<Connection>>
Value <<Attribute>>
ResourceType <<Attribute>>
Requirement <<Entity>>
Resource <<Entity>>
Palette

INSERT connection_mapping.jpg
Figure 5: Realization of Connection Entities


Is Realized as a property of a requirement in the properties view.
Attribute Specification in Metamodel

INSERT attribute_mapping.jpg
Figure 6: Realization of Attributes in AUTODeploy


Is Realized as a Requirement contained by a Component.
Containment Specification in Metamodel

INSERT containment_mapping.jpg

Figure 7: Realization of Containment in AUTODeploy

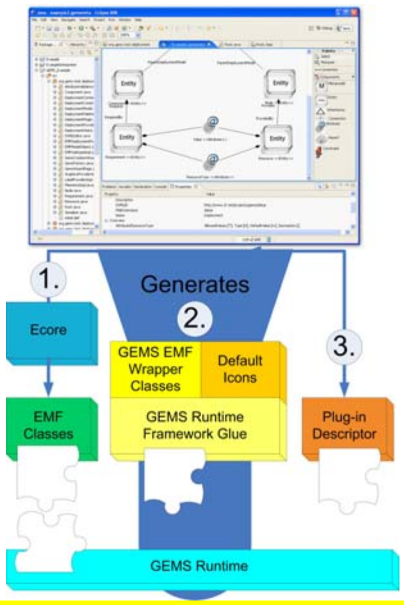
Entities can have *Inheritance* relationships between them, although this example does not show this feature. Inheritance relationships are created by adding an Inheritance element to the model and connecting it to the parent type and derived type(s). Any attributes, connections, or containment relationships that are specified by the parent are inherited by derived types. *Aspects* are another metamodel element that can be defined to group sets of Entities and Connections into separate views on the model. Entity types are associated with an Aspect element in the metamodel through containment relationships. In the generated modeling tool, Aspects appear as unique views that can be used to filter the currently visible Entity and Connection types. Custom CSS styles can be defined for an element so that its visual appearance changes when the aspect changes.

Generating a Graphical Modeling Tool Using GEMS

After creating a metamodel, GEMS's DSML plug-in generator can be invoked to create the modeling tool. The code generator first traverses the various entities in the model and generates an Ecore model, which is a set of EMF objects that represent the metamodel or syntax of the visual language, to implement the metamodel. GEMS then invokes the appropriate EMF code generators to produce EMF classes that implement the Ecore definition. This process can be seen in step 1 of Figure 8.

Comment [ds1]: Jules, can you please briefly explain what Ecore is? – fixed

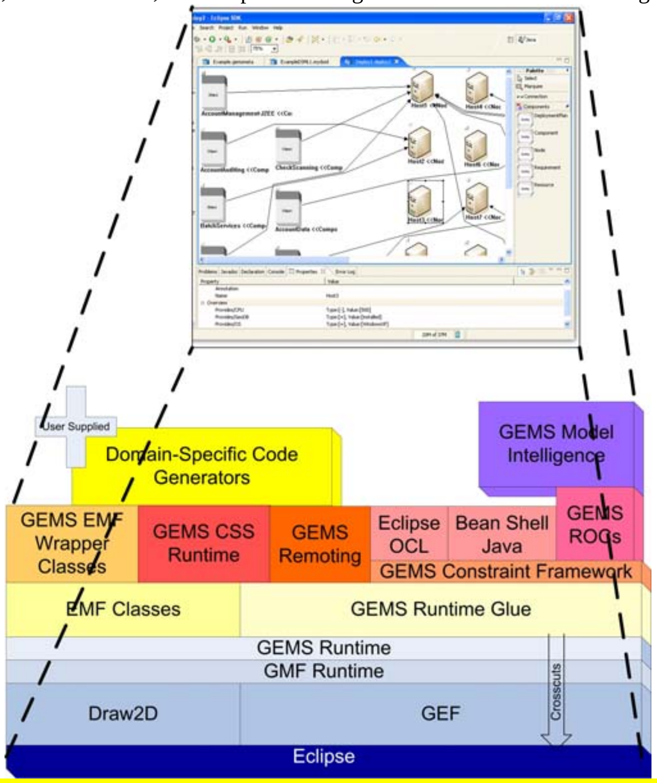
Jules, it's still not really clear what the "EMF metamodel" means/does. Can you please pull this out of the parens and briefly explain it?!



INSERT gems_generation.jpg
Figure 8: Realization of Containment in AUTODeploy

The generated EMF classes are used as the object graph underlying AUTODeploy. These classes automate key aspects of serialization and de-serialization. By default, GEMS leverages EMF's ability to save EMF models to XMI. Other serializers can be plugged in to persist models in a database or use an alternate file format. The EMF object graph also allows GEMS to leverage the libraries available for Eclipse to check Object Constraint Language (OCL) constraints against a model.

The second set of code generated by GEMS are the classes required to plug the generated EMF code into the GEMS runtime framework. The GEMS runtime provides a layer built on top of GEF (and soon GMF) that provides higher level capabilities, such as applying CSS styles to elements, exposing remote update mechanisms, and providing constraint solver modeling intelligence. The relationship between the generated artifacts, GEMS runtime, and Eclipse modeling frameworks can be seen in Figure 9.



INSERT gems_stack.jpg
Figure 9: GEMS and Eclipse Modeling Frameworks

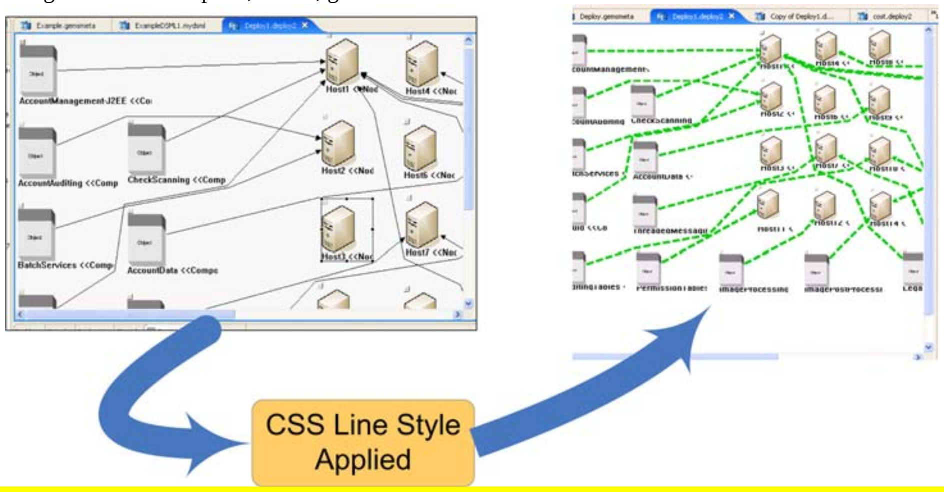
The GEMS runtime provides numerous extension points for adding custom functionality to the generated modeling tool. Extension points are available for adding actions that can be triggered by OCL, Java, or role-based object constraint assertions on the model; customization of menus; custom code generators; remoting mechanisms; custom serializers; and many other features. As the underlying Eclipse modeling technologies evolve, GEMS continues to incorporate and expose their new features.

The third set of code artifacts, generated by GEMS are the various XML descriptors, build specifications, classpath directives, and icons required to integrate the generated tool into Eclipse as a plug-in. When a modeling tool is generated into a Java project, these various artifacts configure the project properly to build the plug-in and make it visible for testing with the runtime workbench. The build artifacts also configure the project so that it can be exported properly as an Eclipse modeling tool plug-in.

Customizing the Look of the Modeling Tool Using GEMS

A goal of GEMS is to avoid requiring developers to use third-generation languages to write any graphics code required to implement a modeling tool. In most cases, however, simply providing a stock set of visualizations will not suffice. GEMS therefore allows developers to customize the look and feel of their modeling tools. GEMS supports this capability by allowing developers to change the default icons visible on the palette and in the model, which enables developers to quickly create impressive visualizations that match the domain notations.

Swapping icons, isn't the only mechanism GEMS provides to customize the look and feel of modeling tools like AUTODeploy. Developers can use CSS style sheets to change fonts, colors, backgrounds, background images, line styles, and many more features of a GEMS-based modeling tool. If an organization has a graphic design department, the customization of the look is similar enough to using HTML and CSS that they can often handle this effort. Figure 10, shows the application of a CSS line style to change the line to a 4 pixel, dotted, green line.



INSERT applying_css.jpg
Figure 10: Applying a CSS Line Style

CSS styles are applied to elements using traditional CSS selectors. The key difference is that the selectors refer to the roles and types specified in the metamodel. For example, to make all the Components that are deployed have a different style than un-deployed Nodes, a style could be created with a selector that matches source role of a Deployment connection. After adding this style, any Component that serves as the source of a Deployment connection will have the style applied.

CSS selectors can also be matched against different aspects of a model. For example, a style can be created to only match the target of a Deployment connection if the current aspect being used for visualization is the Nodes aspect. The Nodes aspect would show only the Nodes in the model and apply the Deployment target style to any node that had Components deployed to it.

Even more complex visual behaviors can be created by leveraging a feature called TAGS in GEMS. TAGS are textual markup that can be added to modeling elements to denote that they have a certain property. For example, AUTODeploy could provide visual queues to modelers by changing the background color of nodes who cannot host any more components to red. Two pieces of code are required to create this complex behaviour based on domain analysis:

1. An OCL, Java, or role-based object constraint trigger must be added to add a "NodeFull" tag to nodes whose resources cannot support any more Components.

2. Create a CSS style that matches Nodes with the "NodeFull" tag.

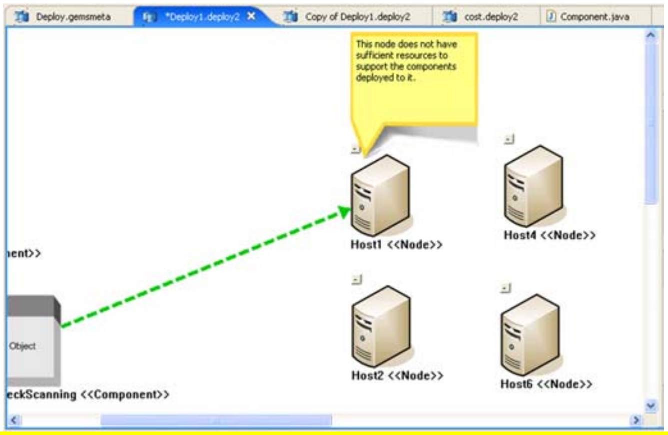
The GEMS TAGS facility supports the combination of domain analysis with CSS styles to rapidly develop complex domain-specific visual behaviors, such as changing the icon for a node that doesn't have sufficient resources to host anymore components or highlighting green nodes that have more than a predefined level of slack on their CPU.

Adding Domain Constraints

A key benefit of MDE tools is their ability to capture and enforce domain constraints. The GEMS constraint framework supports a number of constraint types, including OCL, Java, and role-based object constraints. Other constraint languages can be plugged into GEMS by extending various extension points in the plug-in.

GEMS views constraints more as triggers than actual constraints since this allows constraints to perform a number of actions to guarantee model correctness, such as showing a warning message, vetoing a model change, or adding elements to fix the error. In GEMS, a domain analysis is performed using one of the constraint languages, such as OCL, and when the analysis indicates that a constraint has been violated, an action is triggered. For a traditional constraint, the action can veto the modeling event that produced the constraint violation and roll back any changes that it caused. A trigger can also result in a message popping up above the modeling element that has violated the constraint. Much more complex actions can take place as well. For example, a constraint violation can trigger an action that runs automated diagnostics to identify why the event was generated, as seen in Figure 11.

Comment [s2]: Jules, please briefly explain the motivation for this perspective!



This node does not have sufficient resources to support the components deployed to it.

INSERT constraint_violation.jpg

Figure 11: A Constraint Violation Triggering a Popup Warning

Model Intelligence

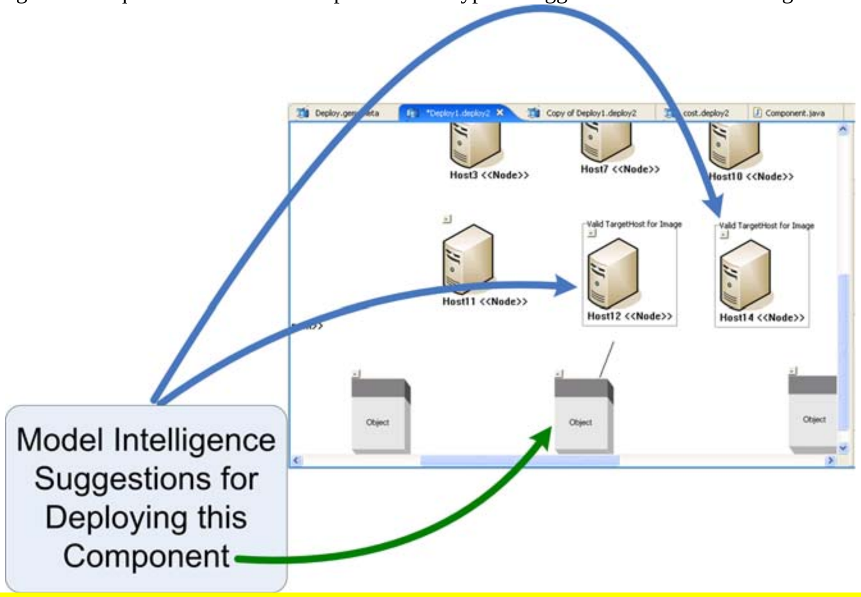
A unique feature of GEMS that sets it apart from other MDE tools is its integration of Role-based Object Constraints (ROCs). ROCs allow developers to describe declarative domain constraints, just like they would with OCL, except that GEMS can leverage its built-in constraint solvers to find solutions for many types of constraint problems. To put it another way, if you tell the GEMS-based modeling tool about what the rules are for a correct solution, it can find the solution for you.

For example, in AUTODeploy, each Component must be deployed to a node that supports the proper configuration of middleware, OS, OS version, databases, etc. These constraints are modeled as

Requirements that must be matched with >,<, or = against a value on a Resource contained by a Node. For example, you could have the requirements, OSVersion > 2.3, MultiCoreCPU = true, FireWallInstalled = true, or HeatProduction < 25. In AUTODeploy, users can add arbitrary name, comparator, value requirements that must be matched against the Resources contained by a Node.

A realistic deployment model for a data center will likely contain hundreds of servers and hundreds or thousands of software components. Human modelers clearly can't point and click their way through a deployment of this size. The GEMS Model Intelligence and ROCs are designed to handle problems at this scale.

Model Intelligence provides a modeling tool like AUTODeploy with two key capabilities. First, it allows the tool to show a modeler the valid ways of satisfying a constraint. For example, in AUTODeploy, when a user places the mouse over a Component, Model Intelligence can highlight the Nodes that satisfy the configuration requirements for the Component. This type of suggestion can be seen in Figure 12.



INSERT local_guidance.jpg
Figure 12: Model Intelligence Suggesting Deployment Locations

Suggesting local modeling decisions is certainly a huge improvement over merely constraint checking a manually produced solution. The real benefit of Model Intelligence becomes evident when it is combined with batch processing. As described previously, large-scale models are too complex to manage manually, particularly if the model has complex global constraints, such as resource constraints. GEMS provides a facility for combining batch processing with Model Intelligence to perform complex assignments automatically, such as finding a target host for EVERY Component in a model.

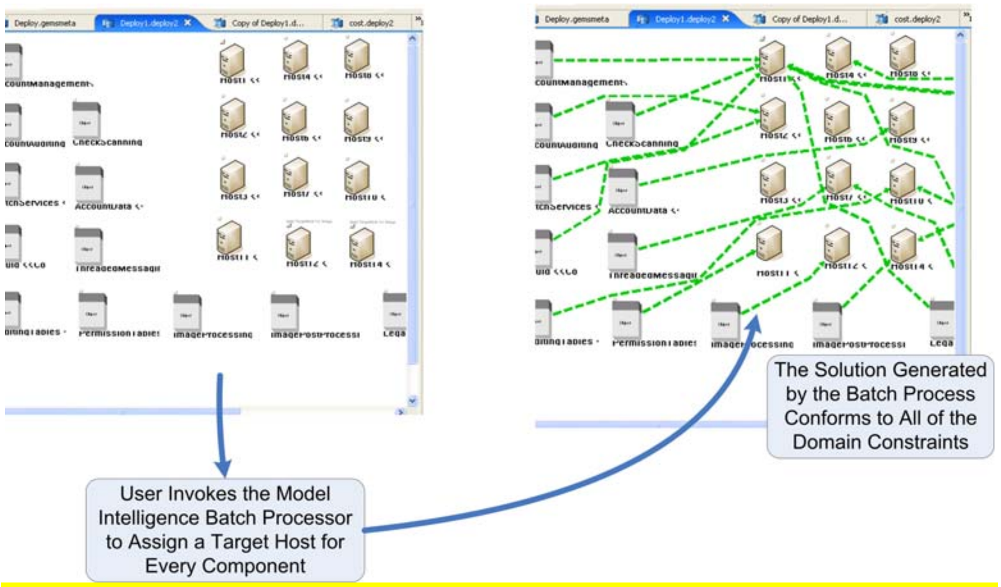
To see how this could work using GEMS, we will add the following ROC rules written in Prolog to AUTODeploy:

```
compare_value(V1,V2,'>') :- V1 > V2.
compare_value(V1,V2,'<') :- V1 < V2.
compare_value(V1,V1,'=').

matches_resource(Req,Resources) :-
  member(Res,Resources),
  self_name(Req,RName),
  self_name(Res,RName),
  self_type(Req,Type),
  self_value(Req,Rqv),
  self_value(Res,Rsv),
  comparevalue(Rsv,Rqv,Type).

can_deploy_to(Componentid,Nodeid) :-
  self_type(Componentid,component),
  self_type(Nodeid,node),
  self_requires(Componentid,Requirements),
  self_provides(Nodeid,Resources),
  forall(member(Req,Requirements),matches_resource(Req,Resources)).
```

These rules implement exactly the requirement to resource matching that we described earlier. They tell Model Intelligence to only deploy a Component to a Node if all of the Requirements are met by the Node. We can then make the rule “can_deploy_to” invokable as a batch process. The result of running this Model Intelligence batch processor can be seen in Figure 13.


User Invokes the Model Intelligence Batch Processor to Assign a Target Host for Every Component
The Solution Generated by the Batch Process Conforms to All of the Domain Constraints
Deploy.gemsmeta
Deploy1.deploy2
Copy of Deploy1.d...
cost.deploy2

INSERT batch_process.jpg
Figure 13: Model Intelligence Assigning a Target Host for Every Component

The ROCs that Model Intelligence uses are a programming paradigm built on top of constraint solvers. The default solver that is packaged with GEMS uses Prolog since it provides a large number of constraint solver and optimizer implementations, an easy to use declarative language, and good performance. ROCs allow users to describe constraints using domain-specific notation, while at the same time allowing existing complex Prolog solvers and algorithms to be plugged-in. ROCs is an extensible paradigm not limited solely to Prolog-based solvers. New solvers, such as highly optimized bin-packers, simplex methods, or other libraries can be incorporated into the paradigm. Plugging a solver into ROCs is a non-trivial task, however, and is rarely needed. In the software component deployment modeling tool that we developed with GEMS, complex Prolog-based solvers have been developed for resource constraints. We are in the process of templatizing and incorporating these complex solvers into the base ROCs installation.

Concluding Remarks

This article has described the powerful features available for building graphical modeling tools with GEMS. The article has just scratched the surface of what GEMS provides and many features, such as remote updating and model templatization have been omitted, due to space limitations. For more information on GEMS, please see our project website [1].

Resources

[1] GEMS: http://www.eclipse.org/gmt/gems

[2] GEMS Download and Development: http://www.sf.net/projects/gems

[3] GEMS Research at the Distributed Object Computing Group http://www.dre.vanderbilt.edu/~jules/gems.htm

[4] The Distributed Object Computing Group http://www.dre.vanderbilt.edu

[5] Model-Integrated Computing at ISIS: http://www.isis.vanderbilt.edu/mic.html

[6] Generic Modeling Environment (GME): http://www.isis.vanderbilt.edu/Projects/gme