PRODUCT LINE PARADIGM FOR ROBOTIC SYSTEM SOFTWARE

William Smuda, PhD
US Army
TARDEC

Matthew Skalny
US Army
TARDEC

Leif Magowan
US Army
TARDEC

ABSTRACT

Software systems, like physical systems, require explicit architectural descriptions to increase system level comprehension. Developing networked robotic systems of diverse physical assets is a continuing challenge to developers. Problems often multiply when adding new hardware/software artifacts or when reconfiguring existing systems. This work describes a method to create reconfigurable software for Army robotic systems via model-based, graphical domain-specific languages and reusable components. The paradigm makes use of feature models, the basis for product line software that describes and constrains variable aspects of the robotic system's hardware and associated software. Domain-specific languages use terms understandable to domain engineers and technicians. Users of the modeling environment are able to stay at a high level of abstraction and need not concern themselves with the details of the composed and generated code.

INTRODUCTION

"Robotics" is a word that infers different meanings to different people, and also to the same person at different times. In this paper we will use the word "robotics" abstractly to infer to a collection of activities necessary to create a useful robot. These activities encompass the entire life cycle of the robot. The collection of activities can be highly structured, structured within a specific phase, ad-hoc or any combination of above. Structured activities are the basis of Systems Engineering. The Product Line Paradigm provides a framework for a set of interconnected structured activities across the life cycle.

Robots, as many other modern devices are collections of embedded systems. Today's cars, trucks and construction equipment are also collections of embedded systems. Most modern automobiles have 20 to 70 electronic control units with 100 to 1000 million object code instructions.[1] Software provides much of the value added to the devices we use day to day, with some estimates of 80% or more in the automotive industry. If we consider commercial vehicles as a starting point, we can see that the number of software instructions can be immense as seen in **Figure 1:** System Size vs Deployment (after 1). Aircraft navigation systems are another good example of a baseline system that requires large efforts for on-board software development, as well as ground support systems. If we consider a commercial air platform as our baseline, we begin to understand the problem of adding specific mission package software. If we consider a commercial truck as our base platform, we may argue that the problem is of an even greater magnitude. There is no existing ground support structure for autonomous vehicles, while airplanes enjoy a "little airplane, big sky" environment. Autonomous ground vehicles are required to operate in a dynamic, cluttered environment.

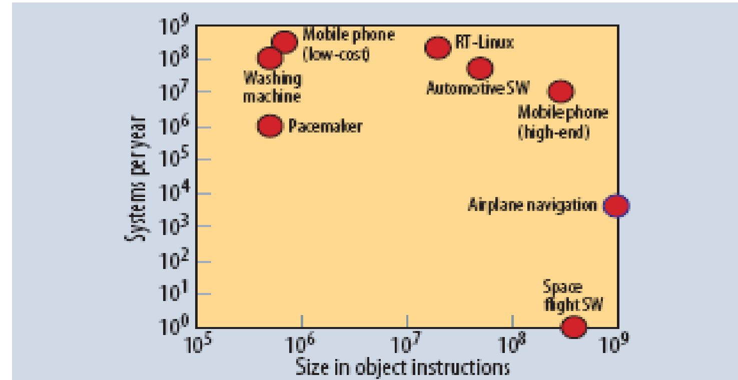


Figure 1: System Size vs Deployment (after 1)

The cost of building effective embedded software is not directly related to the number of units produced. It takes approximately the same effort to produce software for a single system as it does for a fleet. Robotics has all the issues that current commercial systems have. Safety is of course a major concern, since the human is not the prime

controller. Reliability, security, computational resources and heterogeneity, as well as real-time concerns all also play a large role. Furthermore, robotic system may be modified over time, particularly in military systems. The software needs to be accessible and understandable.

Department of Defense (DoD) robotics practitioners are involved at every stage of the lifecycle from concept to use to repair and upgrades. As noted earlier, robotics is a collection of activities. Ad-hoc development is both expensive and time consuming. Vendor internal procedures are not always well known. In order for DoD robotics practitioners to provide maximum value to the taxpayer, a structured systems engineering approach is necessary. Although this paper focuses on robotic software, we will also touch on other aspects where they are relevant. The approach we suggest is the Product Line paradigm.

Product Lines

Product Lines are a familiar concept for consumer products. They are often used to describe a base platform and available options. Usually, consumers use product line descriptions to buy a car or electronics component with the features they desire. Manufactures use product lines to allow consumer choice at reasonable cost by introducing common components into multiple product lines.

Feature Models

Product lines are succinctly defied using feature models. This modeling technique, introduced about two decades ago[2], has both a graphical notation[4] and a textual notation with associated grammar[3]. Designers need to be cognizant of how features are related and of where new features should appear. At domain analysis time, analysts create a feature model. It is a tool for domain analysis to communicate information between developers and users and, if preserved, a temporal tool to determine what the original developers were thinking during a previous spiral.

Feature models present software developers with a tool, much like an assembly diagram for a mechanical developer. As Czarnecki and Eisenecker state in their chapter on feature modeling in their book, *Generative Programming*[4], "Feature Models provide an abstract, concise and explicit representation of the variability present in the software." One should note that the feature model, like the assembly drawing, is not a full representation of the system; it combines with other models for full system representation. A software feature model may have other diagrams and analysis, such as timing constraints, state transition diagrams and object diagrams.

The standard example of a feature model is a representation of a car (Figure 2):

- Mandatory features: engine and transmission
- Optional features: sunroof
- Alternate features: manual or automatic transmission and
- Or-Features: Electric motor, an internal combustion engine or both (hybrid).

Features are the building blocks used to describe concepts. Features are configurable reusable requirements of a concept. The feature modeling process is a study of variability in domain concepts. The process is continuous and iterative, involving identifying as many Use cases, existing feature models, system requirements and additional UML models as possible to identify potential variability points. It also involves recording all supporting information as features become available to the concept. Addressing variability is a key issue in creating reusable software. Decomposition decisions that address the variability discovered in feature modeling result in software with a high level of reusability.

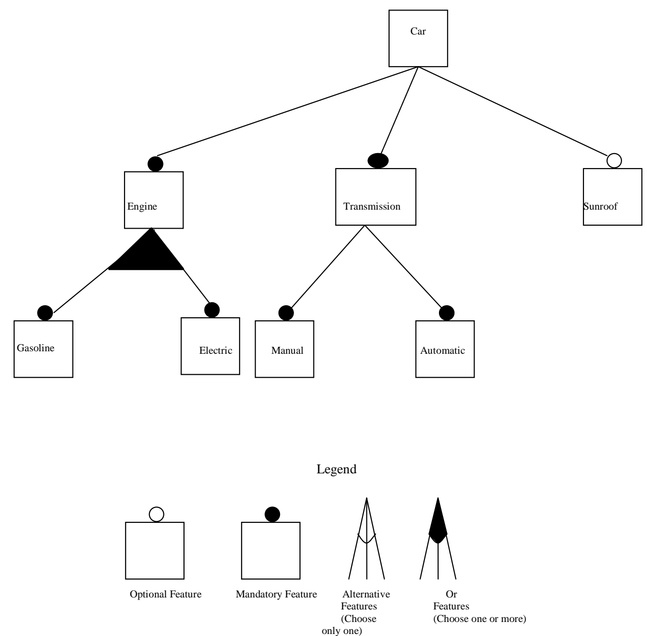


Figure 2: Feature Diagram for a CAR

The feature model is the foundation of the product line paradigm[5]. Formalized feature models can automate segments of the prototyping process. Annotating features

and translating models preserves information, defines data and component storage, and automates many segments of the process. In particular, by automating the integration of concrete realizations of features using components, a prototyping environment can present domain engineers with constrained choices that will greatly simplify the task of assembling software to construct prototype robotic systems.

Feature models contain a great deal of information. The top-level, visible feature diagram, presents an uncluttered view of the concept in all its variants. From an engineering standpoint, this allows analysis of tradeoffs at variability points. Depending on the type of variability, feature models highlight where the system instances are compile time, or run-time-dependent. Additionally, feature models show where concept instances diverge; this may lead to areas where parallel development teams may be able to work without a great deal of coupling. Conversely, feature models also illustrate areas where instances of a concept are common. An important note is that feature diagrams of concepts can be graphs. This indicates that sub-features may be associated with more than one parent feature. The product line can reuse these areas for other instances of a concept or for future extensions to a concept.

Domain Specific Language

Feature models enable Model Driven Engineering. Model-driven Engineering combines Feature models and constraints. Constraints can be part of the Meta-model to guide the domain modeler. Constraints can also be applied when the domain model output is processed to analyze systems or for temporary restrictions. For instance, an automobile model is constrained to a single transmission by noting a cardinality of 1..1 (must have one, must not have more than one) for the transmission model element. A temporary constraint may indicate that a particular model of the engine is not available until some future date.

Domain Specific Languages are described in a Meta-model, often graphical, that defines the relationships of abstractions in the domain. The Meta-model is in essence a feature model. Engineers create the Meta-models in UML, the language of the software engineer, and transform them into a constrained design environment, usually using graphical icons that pictorially describe the abstractions in terms understandable by domain engineers. Domain Specific Languages in turn enable code generation and composition. Software components are constructed to conform to specific design patterns and assembled with Domain-specific Languages.

In essence, a Domain Specific Language is composed of:

- Elements, which typically contain sub-elements and attributes
- Connections and
- Constraints.

A single Domain Specific Languages may be used to generate additional models and analysis. For instance, with proper annotation and repository information, a single instance of a domain model created with a Domain Specific Language has the potential to generate code, generate test cases, compose simulation models, analyze power requirements and build cost estimates.

Tool Support

In recent years, several tools have emerged to support Feature Modeling and Domain Specific Languages. Two that are used in TARDEC IGS are the Generic Modeling Environment[6] (GME) and Eclipse[7].

When TARDEC researchers started this effort, they decided to use tools readily available to DoD engineers and scientists, wanting to avoid the complexity of supporting multiple proprietary tools and environments. The first consideration was to use tools that were developed for the DoD under any of a number of contracts. In particular, they were concerned that the tools should not have a significant initial cost and should not have significant recurring support costs. In particular, researchers wanted to use freely available software with little or no license restrictions for government use. The Generic Modeling Environment (GME) is one of the tools that fit the requirements; funding for GME in part came from the DARPA Information Exploitation Office (DARPA/IXO).

GME is a design environment specifically designed to be configurable to a wide range of domains. GME is configured by creating Meta-models that specify a paradigm for modeling in an application domain. The Meta-models are composed of syntactic, semantic, and presentation information, as well as organization, construction, and constraint information. The paradigm created in the Meta-model defines a family of application-specific models. In the GME Environment, the Meta-model of a specific paradigm is used to automatically gene a target domain-modeling environment.

GME supports a variety of modeling concepts[8] that engineers use to create an architectural description or Meta-model. These concepts include hierarchy, multiple aspects, sets, references and constraints. These concepts, when

composed in a meaningful way, specify software architecture[9].

A GME Meta-model is defined as a project, that has a set of folders to help organize complex models. Folders contain models, which are composed of other models, atoms, references, connections and sets. Models, atoms, references, connections and sets are all GME "First Class Objects" (FCO). The number and kind of FCOs that are allowed in a model is determined by the modeling paradigm under construction and is defined by a containment connection. Contained objects can also be defined with an inheritance relationship. Atoms are elementary objects; they represent the lowest-level element of a model hierarchy. GME objects have attributes associated with the basic concept, such as role, name and kind. GME has a facility where additional attributes can be defined during Meta-modeling. The attributes that can be associated with an object include field (text, integer and double), Boolean and enumerated. If the attributes defined are associated with the parent object in an inheritance hierarchy, then the sub-objects inherit those attributes.

Relationships are modeled by creating a connection between two objects. These connections may be defined as directional or bi-directional. Two objects must have the same parent and be visible within the same aspect. Several kinds of connections can be defined in a single paradigm. The connections determine which objects can participate in a particular relationship, and connections can have attributes and cardinality. If it becomes necessary to associate objects in different parts of the model hierarchy, GME provides a Reference object that can be used exactly as other GME FCO. Any FCO except a connection may be referred to by a Reference.

GME models are similar to classes in Java. They can be sub-typed and instantiated as many times as needed. In order to promote reuse and simplify model maintenance, designers restrict changes that propagate down in the model. Attribute values of model instances can be changes, but no parts can be added or deleted. Sub-typed models may have new parts added, but parts from the parent model cannot be deleted.

GME's Meta-modeling paradigm is based on the Unified Modeling Language (UML). Syntactic definitions are modeled using UML class diagrams, while semantics are specified using the Object Constraint Language (OCL).

There are tools emerging to translate between different environments, specifically between GME and Eclipse[10].

Eclipse offers a widely used and well supported IDE and set of tools. Eclipse is open source and based on plugins, and thus offers a wide degree of flexibility and different projects that can be utilized in a Product Line Paradigm for robotics software. Specifically, the Graphical Modeling Framework (GMF) and Equinox projects offer tools of particular interest that can support a product line paradigm for software product creation.

GMF offers similar capabilities to those described for the Generic Modeling Environment. GMF provides a generative component and rutime infrastructure for developing graphical editors based on the Eclise Modeling Framework (EMF) and the Graphical Editing Framework (GEF)[7,15]. GMF utilizes three primary models – the domain model, the graphical definition model, and the tooling definition model. The domain model definition is similar to the GME meta-model – it utilizes EMF meta models for describing models, and can define relationships between elements in those models. The graphical definition model defines the various graphical figures that will be used to graphically edit a model, such as figures, nodes, and links. The tooling definition model is used to specify palette, creation tools, actions, etc. for the graphical elements used in a model. All three of these models are binded together using a mapping definition model that maps the graphical elements and tools to the domain model. Code can then ultimately be generated from the models created.

Equinox[7] is the Eclipse implementation of the OSGi R4 core framework specification[16]. Current versions of the Eclipse are now built on top of equinox, where features are plugins consisting of multiple OSGi bundles. In the case of a production line approach to the development and deployment of robotics capabilities, these bundles can be seen as the core components that might be stored in a repository. Bundles can include items like serial communications, a sensor adaptor, or a map rendering service. One bundle may have a dependency on other bundles, and can import and export services (i.e. a JAUS communications service). These bundles can also be provisioned to remote OSGi runtimes, which can not only be useful in rapid prototyping type situations, but also potentially for upgrading production line robotic systems quickly in response to rapidly occurring changes in requirements.

There is considerable Product Line Paradigm enabling research conducted under Model-driven Design and Model-driven Architecture. The Object management Group's (OMG) Unified Modeling Language[11] (UML) 2.0 provides increased support. The Generic Modeling Environment from the ISIS center at Vanderbilt University provides a

platform for developing Model-driven designs and the embedded systems community has recognized the power of Model-driven design for developing software product lines for automotive, signal and aerospace applications. The Eclipse Foundation has several projects focusing on Model-driven paradigms.

Rapid Prototyping Example

The design and implementation of software for network systems of diverse physical assets is a continuing challenge to sensor network developers. The problems are often multiplied when adding new elements, and when reconfiguring existing systems. The worst and often typical case is that we need to integrate a collection of artifacts that we can only access via external interfaces. The engineers and scientists are usually robotic or unmanned sensor specialists with a smattering of software knowledge. Experienced software engineers and experienced robotic and unattended sensor engineers with intensive software engineering experience are in short supply.

The objective is to provide a software engineering tool to capture and extend the Software Engineer's knowledge into the realm of the Domain Engineer. This provides consistent high quality prototype development. This research, coupled with well defined object oriented design practices, software reuse and code composition will enable the Product Line Paradigm and hence, system extensibility.

For software systems, like physical systems, explicit architectural descriptions increase system level comprehension. Our research is based on model driven design architecture. High level system models are defined in the Unified Modeling Language (UML), the language of the software engineer. Figure 3 shows a software engineering level meta-model for a family of simple robotic systems. However, since most experimental work is done by non-software specialists, (electronics Engineers, Mechanical Engineers and technicians) the model is translated into a graphical, domain specific model. Figure 4 shows a simple robotic system created from the meta-model in Figure 3. Components are presented as domain specific icons, and constraints from the UML model are propagated into the domain model. Domain specialists manipulate the domain model, which then composes software elements needed at each node to create an aggregate system.

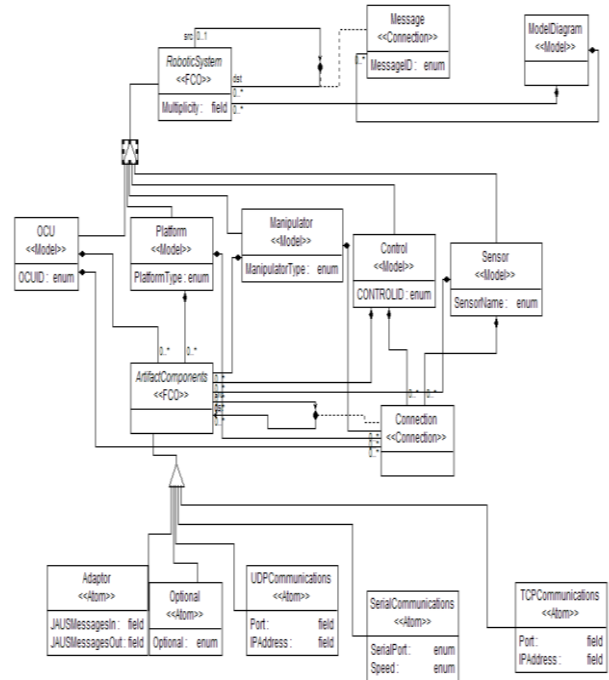


Figure 3: Meta-Model for a Product Line

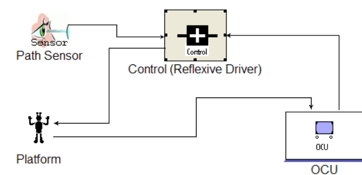
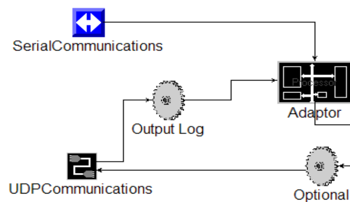
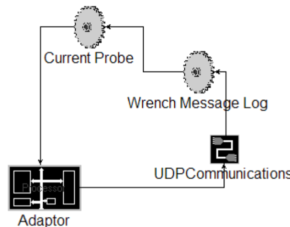


Figure 4: Domain model for Product Line Robot Instance

This particular Domain Specific Language is hierarchical. Each icon in Figure 4 can be opened for configuration. **Figure 5** shows two examples of the items opened for configuration. In each case, there is an adaptor, and at least one communications component. The adaptor converts the legacy protocol of the artifact to and from an XML JAUS format to allow the legacy artifact to interact with the wrapper. The XML JAUS is used within the wrapper to interact with a set of user selectable XML aware instrumentation components. Finally, the communications components convert the XML JAUS format to and from the binary JAUS wire protocol.



Platform Wrapper



Control Wrapper

Figure 5: Domain Engineer adds Adaptor, Communications and Optional Software Components that Configure the Artifact Wrappers

Our overall objective is to develop guidelines, methods and tools to:

- Capture Software Engineering Expertise
- Transfer this knowledge to Domain Engineers
- Capture software elements for reuse
- Capture configuration and execution data and
- Provide tools to simplify the integration process.

Figure 6 illustrates how we propose to incorporate Systems Engineering principles in the prototyping process. In the figure, we find a Use Case diagram representation of the proposed system. A use case is a simple diagram of high-level abstractions. There is little concrete information about implementation. The use case shows how we separate concerns. There are various technical experts, interacting with their own work products. These include a Meta-modeling environment, a domain model, a component repository, and rule set, and the product.

The Meta-modeling environment is the realm of the software engineer in consultation with the robotics domain engineer. The Meta-model is created in GME for this work. The Meta-model is derived directly from a feature model, using UML instead of feature modeling notation. Elements from a feature model are represented as Models if they contain other elements or Atoms if they are leaf nodes.

Domain-specific modeling is the realm of the robotics engineer. The domain model is also a GME artifact. The domain modeling environment is generated from a GME Meta-model. This provides traceability between domain model and its parent Meta-model.

The Component Repository is currently a collection of reusable software components with documentation. It is a collection of software modules that are created by one or more programmers under the direction of the software engineer. For every component, there is an associated documentation file. This file holds instructions on how the module is used, and on what the physical assets are. Physical assets are items such as special cables needed to use the module as well as historical information. In the future, the Repository may grow to include simulation components, material lists or other data.

A rule set is created by the software engineer. It encapsulates software engineering knowledge to operate on the Repository and produce a product. The rule set is triggered by the completed domain model. A completed domain model is exported as XML. The XML is processed by the rule set and the rule set creates the appropriate software to assemble a prototype system.

Rule sets are coupled to a particular Meta-model and hence a Product Line. Rule sets have two main functions:

- parse the domain model and
- create a product.

A particular rule-set will operate with any domain-model instance created by its associated Meta-model.

The product is the realm of the user. At the current state of our research, the product is a set of programs, one for each node in the prototype system. The programs run on computing devices attached to a legacy node, or on a computer platform dedicated to the node.

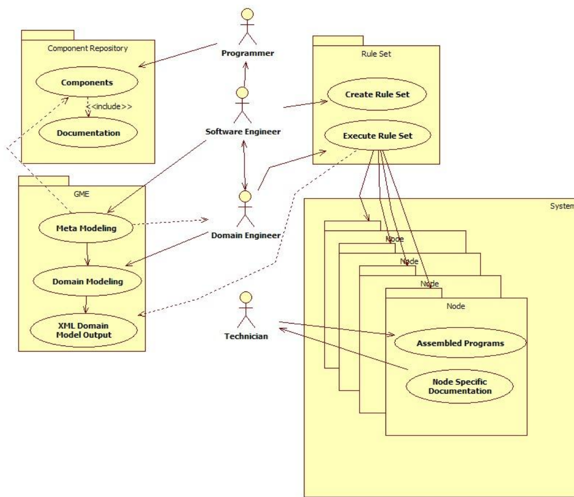


Figure 6: Top Level Design - Use Case

Product Lines in Production Software

Product lines in production software can be defined as a set of software-intensive systems that share a common, managed set of features that meet the requirements of a particular mission and are developed from a common set of core components in a prescribed way[12]. The ultimate goal of product lines in the creation of software products is to increase the efficiency and reduce the cost and time at which software products and groups of related products can be produced, as well as separating the process of developing the core components required for many products from the actual product development itself[13].

There three essential activities for doing product line development for production software: core asset development, product development, and management[14]. Core asset development is the process of developing the capability to produce products. In the context of software production, these core assets will be items such as reusable software components with an associated process for using them within product lines, domain models, requirements specifications, etc. Product development is the actual use of the core assets to develop a software product. Management is required to support both the core asset development and product development.

Production lines for production software are currently used in a variety of domains, including military, medical, automotive, and others[13]. The product line paradigm for production software quite naturally extends into the field of robotics. Elements of production lines can be seen in some specialized robotics software, but they are still mostly setup for the same group of software engineers to do both the development of core assets and production level software. Our focus is on taking this to the next level, and providing a clear separation between the roles of the asset developers (i.e. the software engineer or the programmers seen in figure 6) and the user or domain engineer. This can be done by expanding on the Rapid Prototyping Example provided, and is the focus of our ongoing research.

Conclusions

We believe that the Product Line Paradigm is essential to a Systems Engineering approach to robotic development. In a Systems Engineering approach, there are multiple and diverse stakeholders. These stakeholders need to work together, but should not be required to become expert in each others fields. Emerging tool support facilitates not only the interdisciplinary communications via model translation, but also provides repositories to store intermediate work products. The Product Line Paradigm allows DoD engineers and scientists to abstract vendor processes and products, while enabling analysis and review at all stages of the life cycle.

Disclaimer: Reference herein to any specific commercial company, product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the Department of the Army (DoA). The opinions of the authors expressed herein do not necessarily state or reflect those of the United States Government or the DoA, and shall not be used for advertising or product endorsement purposes.

REFERENCES

[1] C. Ebert and C. Jones, Embedded Software: Facts, Figures, and Future, IEEE Computer, April 2009, pp42-52.

[2] K. Kang, S. Cohen, J. Hess, W. Novak and S. Peterson, Feature-Oriented Domain Analysis (FODA) – Feasibility Study, Report number CMU/SEI-90-TR-21, 1990.

[3] B. Dawes, “Feature Model Diagrams in text and HTML,” http://www.boost.org/more/feature_model_diagrams.htm , 09 April 2007.

[4] K. Czarnecki, U. Eisenecker, “Generative Programming: Methods, Tools, and Applications,” Boston, MA, Addison-Wesley, 2000.

[5] W. Smuda, "Rapid Prototyping of Robotic Systems", Naval Postgraduate School Dissertation, June 2007.

[6] Generic Modeling Environment, http://www.isis.vanderbilt.edu/projects/gme, 09 April 2007.

[7] Eclipse, http://www.eclipse.org/, 09 April 2007.

[8] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, c. C. Thomason, G. Nordstrom, J. Sprinkle, P. Volgyesi, "The Generic Modeling Environment," Proceedings of IEEE WISP'2001, Budapest, Hungary, May 2001.

[9] D. Perry, A. Wolf, "Foundations for the study of Software Architecture," ACM Software Engineering Notes, vol 17, no 4, 1992, pp. 40-52.

[10] J. Bezivin, C. Brunette, R. Chevrel, F. Jouault and I. Kurtev, Bridging the Generic Modeling Environment and the Eclipse Modeling Framework, Proceedings of the Best Practices for Model Driven Software Development at OOPSLA'05, San Diego, California, USA, http://www.softmetaware.com/oopsla2005/bezivin2.pdf

[11] D'Souzaand D.,. Wills A, "Objects Components and Frameworks with UML," Addison Wesley, 1999.

[12] Clements, P. and Northrop, L. "Software Product Lines: Practices and Patterns," Addison-Wesley, 2002.

[13] Chastek, G., Donohoe, P., and McGreggor, J. "A Study of Product Production in Software Product Lines," Carnegie Mellon Univserity Software Engineering Institute, 2004.

[14] "A Framework for Software Product Line Practice", Carnegie Mellon University Software Engineering Institute, http://www.sei.cmu.edu/productlines/framework.html.

[15] GMF Tutorial, http://wiki.eclipse.org/index.php/GMF_Tutorial

[16] OSGi Alliance, http://osgi.org