A Generative and Model Driven Framework for Automated Software Product Generation*

Wei Zhao Barrett R. Bryant
Jeffrey G. Gray Carol C. Burt
Computer and Information Sciences
University of Alabama at Birmingham,
Birmingham, AL 35294-1170, U.S.A.
{zhaow,bryant,gray,cburt}
@cis.uab.edu

Rajeev R. Raje
Andrew M. Olson
Computer and Information Science
Indiana University Purdue University Indianapolis
Indianapolis, IN 46202, U.S.A.
{rraje, aolson}@cs.iupui.edu

Mikhail Auguston
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943, USA
auguston@cs.nps.navy.mil

ABSTRACT
Component-based Software Engineering (CBSE) and related technologies have demonstrated their strength in recent years by increasing development productivity and parts reuse. Recently, the Model Driven Architecture (MDA) has raised the abstraction level of programming languages to modeling languages that can be compiled by downward model transformations. Correspondingly, the goal of Generative Programming (GP) is to automate concrete software product generation from a domain-specification and reusable components. This paper describes the UniFrame framework, which is built on the foundation of CBSE while leveraging the capabilities offered by MDA and GP. UniFrame provides theories and implementation for steps of model transformations for a concrete software product based on domain development in various Generative Domain Models (GDMs).

Keywords
Component-based Software Engineering, Model Driven Architecture, Generative Programming, Domain Engineering, Application Engineering, Two-Level Grammar, Generic Modeling Environment, Feature Modeling.

1. INTRODUCTION
An upward shift in abstraction often leads to an increase in productivity and usually depends highly on the automation of transforming the higher-level abstraction to the lower-level abstractions. As programming languages made their evolution from machine language to assembly language, to 3rd generation languages (FORTRAN, COBOL, C, Java, etc.), programmers were able to concentrate more on the essence (inherited concepts and relationships in applications) of the application rather than being distracted by accidental difficulties (e.g., the constraints and syntax of underlying hardware and technologies) [Bro87]. The trend is that the programming language will ultimately evolve up to the concepts and data set relationships in the problem domain space. This necessitates that a whole framework, rather than a simple conventional compiler, is needed for getting this high level language to be executed by computers directly; at the same time, this high level "language" is not restricted to the traditional sense of language definition[1] but rather a combination of language and tool support. In this paper, we describe our efforts for constructing such a compilation framework and the formal transformation and validation techniques to be integrated into this high level language supporting toolset.

The paper is organized as follows. The Generic Modeling Environment (GME), the modeling tool we used in our research, is briefly mentioned in section 2. Section 3 describes the Two-Level Grammar (TLG), the formal language for specifying the domain models and model transformations. The framework architecture is explained in section 4, and the paper concludes in section 5.

2. GENERIC MODELING ENVIRONMENT
The Generic Modeling Environment (GME) [GME00], developed at the Institute for Software Integrated Systems (ISIS) at Vanderbilt University, is a meta-configurable toolset that supports the easy creation of domain-specific modeling and program synthesis environments. GME provides generic modeling primitives that assist any domain-specific environment designer to create meta-models[2] for domain-specific modeling. The domain experts can use this tailored modeling environment to construct the domain-specific models.

We use the GME for two primary purposes:

1. At the domain engineering level, the GME is used by the domain environment analysts to create domain

* This research is supported by the U. S. Office of Naval Research under the award number N00014-01-1-0746.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

[1] Traditional programming languages are defined by lexical, syntactic and semantic meanings.

[2] The meta-model is also called domain modeling paradigm and environment, or domain modeling concepts and language definition.

feature meta-models and the domain feature models[3]. In the meta-models, the concepts for constructing feature models (e.g., mandatory features, optional features, alternative features, or-features) should be defined using the generic-modeling primitives built into the GME. Feature models [Kan98] describe the common and variable features of the products, their interdependencies, organizations and supplementary information. In other words, feature models are the visualized specifications for the domain where the knowledge of manufacturing the individual products out from the domain is embedded.

2. At the application engineering level, the GME is used to provide the environment for the domain experts (a.k.a. requirements analysts, business analysts) to construct the application model (or requirements model). The application models are constructed using the same domain-specific modeling definitions designed by the domain environment analyst, and also under the context of feature models of this domain. This permits validations and configurations to be checked automatically during the construction. For example, a feature model could be constructed that specifies that a car **transmission** can be either **automatic** or **manual,** but not both. This relationship is called "alternative" [Cza00]. If the domain expert configures the car to have both the automatic and manual transmissions, the violation is checked based on the meta-model since the alternative relationship used in the feature model is defined in the meta-model. But, if the domain expert configures the car transmission to be something called **not-invented-transmission**, then the error can only be checked based on the knowledge from this feature model. The application model is the starting point of our model transformation series.

GME is a means to visualize the domain concepts and concept organization for the environment analyst and to visualize the application organization to the domain experts. However, in order to provide the full capability of configuration validation of applications, and also since GME has become an open source project, we propose to augment it in the following two senses:

1. Being a visual language, the feature model by nature cannot capture the full semantics of logic, constraints, interdependencies of features and Quality of Service (QoS) compositions [Raj02]. We plan to integrate the GME with a formal grammar, Two-Level Grammar (TLG) that is logically computable to specify the visualizable feature model plus the constraints beyond the model [Bry02b].
2. The feature meta-model constructed in the GME only provides restricted environmental checking for the application construction which depends more on the knowledge from domain feature models themselves, e.g. the **not-invented-transmission** error. The feature model specification in TLG can carry the semantics of the feature model from the domain engineering space to the application engineering space, providing the syntax and constraint semantics for the application configuration.

[3] In the GME's terminology, this feature model stands at the modeling level, while in the context of this paper, the feature model is at domain engineering level which serves as the "meta" for application engineering.

3. TWO-LEVEL GRAMMAR

Two-Level Grammar (van Wijngaarden or W-grammar) is an extension of context-free grammars originally developed to define syntax and semantics of programming languages. It was quickly noticed that TLG defines the family of recursively enumerable sets [Sin67], while suitable restrictions yield context-sensitive languages [Bak70]. It has been used to define the complete syntax and static semantics of Algol 68 [Wij74]. Recently it was extended with object orientation, and was developed as an object-oriented requirements specification language integrated with VDM tools for UML modeling and Java and C++ code generation [Bry02a].

The term "two-level" comes from the fact that a set of formal parameters may be defined using a context-free grammar, the possible strings generated from which may then be used as arguments in predicate functions defined using another context-free grammar. From the object-oriented point of view, the set of formal parameters are a set of instance variables and the predicate functions are the methods that manipulate the instance variables. Originally, the first level context-free grammar rules were called the meta-productions or meta-rules, while the second level parameterized context-free grammar rules were called hyper-rules/productions.

The substitution process of the first level grammar is nothing new from that of a regular context free grammar and is called simple substitution; while the essential feature of TLG is the Consistent Substitution or Uniform Replacement in the second level grammar, i.e. an instance of a meta variable must be consistently replaced in a hyper rule [Pem].

e.g. Thing :: letter; rule.

Thing list: Thing; Thing, Thing list.

will generate:

letter list: letter; letter, letter list.

rule list : rule; rule, rule list.

The "::" indicates the meta-level production, and the ":" identifies the hyper level production. Only the nonterminals are allowed in the left side of the meta-level; both the nonterminals and terminals can appear in the left side of the hyper production, and the right side of both meta and hyper productions. Nonterminals are with the first letter capitalized, and terminals are all in lower case letters. ";" is for the "or", and "," is for the "and".

The two levels of TLG make it very convenient to specify the feature models. The first level is used for specifying the feature organization, and the second level is used for specifying the things that are beyond the pure organization, such as feature attributes, relationship cardinalities, pre and post condition for the configurations, interdependencies among the features (the relationship beyond the direct parent and children features).

In order to demonstrate, we present the following artificial example.

(keywords are in bold face).

Class Automobile.
(1) Automobile :: CarBody , Transmission , Engine , Tires .
(2) Transmission :: automatic ; manual .
(3) Engine :: electronic ; gasoline; electronic , gasoline.
…….
(4) Type : car
(5) Automobile **derive** Tires : **if** Automobile.Type = car, Automobile #1, Tires #4;
if Automobile.Type = truck, Automobile #1, Tires #8.
(6) some-post-conditions Transmission :: Transmission some-pre-conditions.
………
end class

In this simple code:

(1): An automobile has 4 parts: car body, transmission, engine and tires.

(2): The transmission can be either automatic or manual.

(3): The engine can be either electronic or gasoline, or both.

The above is the first level context-free grammar.

(4): The "Type" is not one of the nonterminals in the meta-level, so it stands as the attribute for this root class, which is "Automobile". That "Type" derives "car" simply means "car" is "Type" 's value.

(5): "**derive**" , "**if**", **"="** and **"#"** are generic keywords. Generic keywords are built in the TLG, and keywords for domain-specific relationships, configurations, and constraints are defined and derived automatically from the domain meta feature models. This statement refers to the production where the "Automobile" can derive the "Tires", and it represents the cardinality of the configuration between the automobile and the tires. "Automobile.Type" has the object-orientation flavor.

(6): By the consistent substitution rule, the second transmission needs to be substituted by the string generated from the "Transmission" in the meta-level grammar. So, this says in the statement (2), only if both the pre-condition with (2)'s right hand side and the post-condition with (2)'s left hand side are satisfying, then the final configuration process for (2) can be completed.

We can get the meta-level and part of the hyper-level grammar by automatically transforming the feature models. The transformation rules can be built into the GME tool. Part of the hyper-level grammar that is beyond the feature model can be obtained by GUI input. Just based on a TLG interpreter (a little more than a simple parser), the product configuration and validation can be highly automated. Also since both the meta and hyper level grammar are context-free grammars, the construction of this TLG interpreter can be facilitated by the existing parser generators, such as CUP [CUP99].

4. ARCHITECTURE OVERVIEW OF UNIFRAME

The UniFrame project [Raj01][UniFr] is a framework for providing architecture for automated software product generation, upon an order requirement, based on the assembly of a selection from an ensemble of searched software components.

4.1 Fundamental Theses of this Framework

4.1.1 Component-based software engineering

The implementation of UniFrame is built upon the maturity of component-based software engineering [Hei01] because the application generators dynamically configure the application out of a set of available components based on their configuration rules and dependencies embedded in the GDM. In our framework, features are components. The separation of reusable feature (asset) development in the domain engineering and the product configuration using those assets in application engineering reflect the fundamental discipline of the separation of component development and component composition, and hopefully ultimately leads to a component market.

4.1.2 Software development paradigm shift: from single application development to system family development

System family engineering is also called Generative Programming [Cza00] and Product-line Engineering [Wei99]. Domain Engineering is the activity of collecting, organizing, and storing past experience in building systems or parts of systems in a particular domain in the form of reusable assets, and the application engineering is the process of producing concrete systems using the reusable assets developed during domain engineering. In [Cza00], the authors offered a notion of Generative Domain Model (GDM), which is the result of the domain engineering consisting of the feature models, and the notions that are beyond the feature models such as configuration constraints, test plans, feature implementations, QoS calculations, domain prototypes, etc. This concept of paradigm shift is the core design of the UniFrame.

4.1.3 Capture, formalism, modeling and reuse of engineering knowledge

Any software system has the domain-specific concepts and logic, has its structure and its implementation in some concrete technologies. Decisions made on how to produce the software using those concepts comprise the engineering knowledge. In current software engineering practice (single system development), the engineering knowledge is scattered among the policies from domain business executives, expertise from domain experts, experiences from software managers and engineers, and the techniques from software developers and programmers. During the software production process, these decisions will contribute respectively towards the goal of the system, detailed business logic of the system, specifications of software architecture and role assignments for developers, concrete software development by applying different programming languages and component-based technologies.

However, when we move the development paradigm to the product-line, with the goal of manufacturing the concrete software product from the GDM automatically, the engineering knowledge specific to that end product must be formally defined to guide this automation. Toward this end, we categorize the engineering knowledge clearly and formally into three domains [Zha02b]:

1) Business domains are associated with the natural categorization of business sectors and the natural hierarchical structure of business organizations;

2) Architecture domains can be seen as a set of reference architectures or software patterns, which identify the functionality, the role and the collaboration means among different parts of software; and

3) Technology domains address the issues related to software implementation technologies such as component models, programming languages, hardware platforms, and so on.

In order to automate the concrete software generation, we need to perform the domain engineering on the three dimensions of engineering knowledge. We refer to the GDM for each of the dimensions as Business GDM, Architecture GDM, and Technology GDM.

4.2 Framework structure

4.2.1 Domain Level Development

As can be seen in the Figure 1 (see end of paper), domain level engineering consists of three pieces of independent domain development: business domains, architecture domains and technology domains. We use GME to construct feature models in the business GDM and architecture GDM, and those models are translated into TLG internally inside the future augmented GME. The architecture GDM specifies the commonality, variability, and configuration for software patterns. At our current research stage, the technology GDM is only concerned about the technology mapping for the interoperability among heterogeneous software components. The translated TLG can provide a means for early prototyping in the domain, and set the context for the application development as well.

Features should be standardized in each domain and are continually evolving as the domain requirements, which are different from application requirements, evolve.

In each domain, domain asset developers are producing domain-specific features and other artifacts such as test plans, manuals, tutorials, maintenance, etc. These features are component-based and are designed for reuse. Along with the implementation for the features, the developer should provide a Unified Meta-Model for this feature (UMM[4]) [Raj00] so that in the application engineering phase, the generator can use the UMM to identify the feature in the GDM and calculate QoS measurements of the system. If the domain is large enough, a set of available features are not limited to reside on one computer, one network or one organization, they will be dispersed over the Internet and across the organization structures. Features are registered to the UniFrame system for later discovery by the UniFrame Resource Discovery System (URDS) [Sir02].

4.2.2 Application Level Development

In the application engineering phase, we perform a series of model transformations starting at the requirements model and ending at the concrete product. Requirements analysts construct the requirements model in the GME under the context of feature models of this domain. This requirements model needs to be translated into the TLG model for a complete validation. The mapping is two-way, the changed and corrected TLG model should also be re-visualized in GME. The same process applies for the architecture model.

The requirements model, an instance configuration of the business GDM, gets transformed into an instance configuration of an architecture, and any instrumentation code specific to this architecture will be generated at this time.

With the knowledge of the product requirement, and the individual parts in that architecture, the system goes out to the Internet and looks for the necessary features implementation in the business domain using the URDS discovery system. If there are any inconsistencies in the technologies used in those features, the system will generate glue/wrapper code based on the knowledge from the technology GDM [Zha02a].

Then the system QoS validation [Sun02] and final assembly process are carried out automatically by using the information in the three GDMs and UMM associated with each feature implementation.

The GDMs, the requirements models, the application architecture models and UMMs are all internally represented in TLG which acts as the transformation engine.

5. RELATED WORKS AND CONCLUSION

Recent research efforts such as MDA, Generative Programming and Product line Architecture have the same characteristic that is moving the development abstraction one level up. The framework described in this paper bridges the gap between MDA and the System Family Development Paradigm by providing detailed steps of model transformations based on the result of system family engineering.

This paper also serves as the research effort contribution for the open source MDA project that we are affiliated with [MDA02]. In MDA terminology [Fra03], automatic transformations are processed from the Platform Independent Model to Platform Specific Models. In this paper, we explicitly give three stages of transformations, i.e. the variations of "platform". The knowledge in three GDMs provides the meta-information about the various platforms and the rules for steps of refinement (the rules are still being researched).

6. REFERENCES

[Bak70] J. L. Baker, *Some Formal Properties of the Syntax of ALGOL 68*, Doctoral Dissertation, University of Washington, 1970.

[Bro87] F. P. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering." *Computer*, Vol. 20, No. 4, pp.10-19, 1987.

[Bry02a] B. R. Bryant, B.-S. Lee, "Two–Level Grammar as an Object-Oriented Requirements Specification Language," *Proc. 35th Hawaii Int. Conf. System Sciences (HICSS)*, 2002. http://www.hicss.hawaii.edu/HICSS_35/HICSSpapers/PDFdocuments/STDSL01.pdf

[Bry02b] B. R. Bryant, M. Auguston, R. R. Raje, C. C. Burt, A. M. Olson, "Formal Specification of Generative

[4] Briefly, UMM is used to specify the reusable components by providing the values for numerous parameters in the three GDMs.

Component Assembly Using Two-Level Grammar," Proc. SEKE 2002, 14th Int. Conf. Software Engineering and Knowledge Engineering, pp. 209-212, 2002.

[CUP99] CUP Parser Generator for Java. http://www.cs.princeton.edu/~appel/modern/java/CUP/

[Cza00] K. Czarnecki, U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.

[Fra03] D. S. Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing.* Wiley Publishing, Inc., 2003.

[GME00] GME User's Manual. *The Institute for Software Integrated Systems, Vanderbilt University.* http://www.isis.vanderbilt.edu/Projects/gme/Doc.html

[Hei01] G. T. Heineman, W. T. Councill, *Component-Based Software Engineering: Putting the Pieces Together.* Addison-Wesley, 2001.

[Kan98] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, M. Huh, "FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures," *Annals of Software Engineering* 5, pp. 143-168, 1998.

[MDA02] An Open Source MDA project. *OOPSLA 2002 Workshop on Generative Techniques in the Context of Model Driven Architecture* http://www.softmetaware.com/oopsla2002/positionstatement.html#Proposal

[Pem] S. Pemberton, "Executable Semantic Definition of Programming Languages Using Two-level Grammars (Van Wijngaarden Grammars)." http://www.cwi.nl/~steven/vw.html

[Raj00] R. R. Raje, "UMM: Unified Meta-object Model for Open Distributed Systems," *Proc. ICA3PP 2000, 4th IEEE Int. Conf. Algorithms and Architecture for Parallel Processing,* 2000, pp. 454-465.

[Raj01] R. R. Raje, M. Auguston, B. R. Bryant, A. M. Olson, C. C. Burt, "A Unified Approach for the Integration of Distributed Heterogeneous Software Components," Proc. 2001 Monterey Workshop Engineering Automation for Software Intensive System Integration, pp. 109-119, 2001.

[Raj02] R. R. Raje, M. Auguston, B. R. Bryant, A. M. Olson, C. C. Burt, "A Quality of Service-Based Framework for Creating Distributed Heterogeneous Software Components," *Concurrency and Computation: Practice and Experience* 14 pp. 1009-1034, 2002.

[Sin67] M. Sintzoff, "Existence of van Wijingaarden's Syntax for Every Recursively Enumerable Set," *Ann. Soc. Sci. Bruxelles 2,* pp. 115-118, 1976.

[Sir02] N. N. Siram, R. R. Raje, B. R. Bryant, A. M. Olson, M. Auguston, C. C. Burt, "An Architecture for the UniFrame Resource Discovery Service Proc. SEM 2002, 3rd Int. Workshop Software Engineering and Middleware, Springer-Verlag Lecture Notes in Computer Science, Vol. 2596, 2002.

[Sun02] C. Sun, R. R. Raje, A. M. Olson, B. R. Bryant, M. Auguston, C. C. Burt, Z. Huang, "Composition and Decomposition of Quality of Service Parameters in Distributed Component-Based Systems," *Proc. of Fifth IEEE Int. Conf. Algorithms and Architectures for Parallel Processing*, pp. 273-277, 2002.

[UniFr] UniFrame Project http://www.cs.iupui.edu/uniFrame/

[Wei99] D. M. Weiss and C. T. R. Lei, *Software Product-line Engineering: A Family-Based Software Development Process.* Addison-Wesley, 1999.

[Wij74] A. van Wijngaarden, "Revised Report on the Algorithmic Language ALGOL 68." *Acta Informatica*, 5, pp. 1-236, 1974.

[Zha02a] W. Zhao, "Two-Level Grammar as the Formalism for Middleware Generation in Internet Component Broker Organizations," *Proceedings of GCSE/SAIG Young Researchers Workshop*, held in conjunction with the *First ACM SIGPLAN Conference on Generative Programming and Component Engineering*, 2002. http://www.cs.uni-essen.de/dawis/conferences/GCSE_SAIG_YRW2002/submissions/final/Zhao.pdf

[Zha02b] W. Zhao, B. R. Bryant, F. Cao, R. R. Raje, M. Auguston, A. M. Olson, C. C. Burt. "A Component Assembly Architecture with Two-Level Grammar Infrastructure". *Proc. of OOPSLA'2002 Workshop Generative Techniques in the Context of Model Driven Architecture,* 2002. http://www.softmetaware.com/oopsla2002/zhaow.pdf

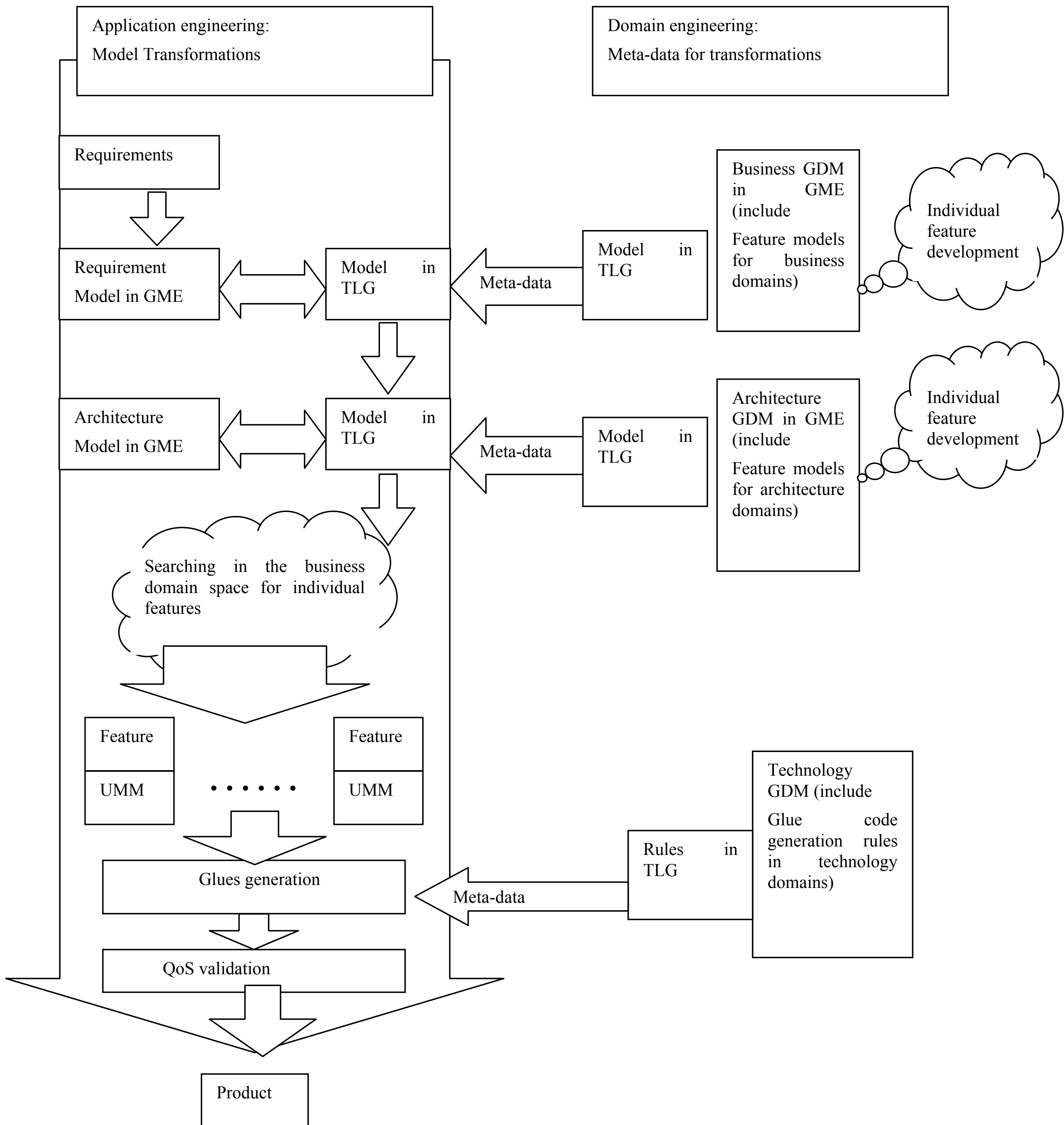

Figure 1. The UniFrame System Structure