Two-Level Grammar as the Formalism for Middleware Generation in Internet Component Broker Organizations [*]

Wei Zhao[**]

Abstract

During the software production of any business domain, we will encounter components coming from different component models. Realizing the interoperability among heterogeneous component models at technology domain level is one of the fundamental difficulties of achieving product line constructions at business domain level. Our research of automatic glue and wrapper code generation to compose the components adhering to different component models follows the idea of Generative Programming (GP) [Cza00]. The Generative Domain Model (GDM) for glue/wrapper code production consists of two levels: 1) meta-specifications for heterogeneous components from the component level, 2) on the system level, there are component model domain-specific Knowledge Bases (ds-KBs) together with their associated middleware[1] code generators. Two-Level Grammar is the formalism for both the component and system level specifications, thus is the key technology to automate middleware code generation.

1. Introduction and Architectural overview.

Generating a concrete software product from domain feature models still remains a dream in today's software engineering. In order to automate this manufacturing process, one needs to be able to capture and model the various aspects of engineering knowledge for any possible software solution for a particular domain. And this knowledge would be a synthesis 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. Our study has categorized this independent (in terms of knowledge origins), fluid (involved in many or all the phases of software development), and interrelated (one will affect another horizontally and/or vertically) synergy into three domains [Zha02]: 1) Business differentiated domain, the meaning of domain is associated with natural categorization of business sectors in the real world; 2) Functionality differentiated domains, based on the functionality of different parts of the software, e.g. software components might be developed for content presentation, business logic computation or resource access and so on; and 3) Technology-differentiated domains, which vary according to the various developing technologies such component models and programming languages.

Should we successfully construct the Generative Domain Model (GDM) [Cza00] for each domain mentioned above, we would be able to automate the ultimate software products under the guidance of how and when the interaction among the three GDMs will take place.

As an alliance to the Model Driven Architecture (MDA) [OMG01] direction from Object Management Group (OMG) community, we expect that for any business domain, there will be an Internet Component Exchange and Assembly (ICEA) center specialized at researching and developing the Generative Domain Model (GDM) for its business domain, and some organizations designated as Internet Component Developers (ICD) responsible for translation from platform specific models (PSMs) to executable implementations [Bur02].Quality of Services (QoS) issues in component assembly process are of special interest in our research although not especially covered in this paper. Thereby some official organizations for certifying and insuring the QoS provided by the individual components along with their associated service price, and the predicted QoS of the system after composition will act an important role in the future software component assembly and trading market.

The topic of this paper concerns the Internet Component Broker (ICB) [Raj01] organizations, and this conforms to the technology domain

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

[**] Department of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, AL 35294-1170, U.S.A., zhaow@*cis.uab.edu.*

[1] In this paper, middleware has the same meaning as glue/wrapper.

aforementioned and is part of our full research. ICBs develop and maintain the KBs for component models and technologies and are responsible for developing the necessary technology parameters for the mapping from platform independent models (PIMs) to PSMs. The ICB serves all the business domains for building the multi-model interoperability for their business components. ICB is analogous to Object Request Broker (ORB). As opposed to provide the capability to generate the glue and wrapper necessary for objects written in different programming language to communicate transparently, the ICB provides the interoperation for components implemented in diverse component models and thus presents a collaboration vision one level above the ORB.

For a complete architecture explanation, readers are referred to [Zha02].

A Unified Meta-component Model (UMM) [Raj00] comprises the component level specification of the GDM for ICB organizations and will be delivered from ICEAs along with their brokering requests. UMM formally and uniformly represents the computational, cooperative, economic and deployment knowledge and requirements of the distributed and heterogeneous software components, explicitly mocking up the semantics of the services a component carries. UMM computational aspects include the lexical, syntactic and semantic meaning of the components. The lexical meaning is the business domain specific naming of functions and QoS domain specific parameters; the syntactic contract is comprised of the component literal functionality interfaces; what the functionality and service this component provides tells the semantic view of the component. The semantics of the component is the most critical joint toward the great success of ultimate component exchange and assembly. The Uniform Resource Identifiers (URI) located in their ICEA uniquely identifies the standardized services for each business domain. COTS components are required to cooperate with each other, and the UMM cooperative aspect takes care of the interrelationship among the components such as expected collaborators. The economic aspect includes the QoS provided by this component, associated price and trading policies, and so on. Some deployment issues such as component model and technologies used, operating system platforms, underlying network quality, etc. constitute the deployment aspect of the UMM. UMM is formally represented in TLG classes as shown in examples in section 2, mainly concentrating on computational and deployment aspect views of UMM.

At the system level, we only cover the generator specification.

2. Two-Level Grammar (TLG).

Two-Level Grammar (van Wijngaarden or W-grammar) is an extension of the context-free grammar originally developed to define syntax and semantics of programming language. 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 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, C++ code generations [Bry02].

The term "two-level" comes from the fact that a set of formal parameters may be defined using a context-free grammar, the possible string 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 was called the meta-productions or meta-rules, while the second level parameterized context-free grammar was called hyper-rules/productions.

When TLG is restricted and adapted to fit in the goal of component specification and code generation, we formally defined it as a triple (modified from [Gre74] and [Saa89]):

$\mathbf{G=[(S,E), (V_m,P_m),(V_h,P_h,s)]}$, where:

§ S is a finite set of terminals consists of syntactic markers and TLG built-in vocabulary appearing as lower case English words and literal symbols.

§ **E** is a finite set of external terminals such as customer language syntax markers, code fragments, etc. A finite set of $(E-V_h)$ will be enclosed in "", otherwise will be the same form as Σ.

§ $\mathbf{(V_m,P_m)}$ is called the meta-level and consists of:

♠$\mathbf{V_m}$ is a finite set of meta-nonterminals, for which $V_m \cap E \cap \Sigma = \Phi$, V_m is written in English words with the first letter capitalized, and

♠$\mathbf{P_m}$ is a finite set of meta productions.

§ $(\mathbf{V_h}, \mathbf{P_h}, s)$ is called hyper-level and consists of:

♠ $(\mathbf{V_h} - S)$ is a set of hyper variables, $V_m \subset V_h$, $E \cap V_h$? Φ, which means all the internal and external terminals and meta variables can be part of hyper notion ($\mathbf{V_h}$) appearing in both sides of hyper rules[2].

♠$\mathbf{P_h}$ is a set of hyper productions, and

♠ $s \subset V_h - \Sigma$, start symbol.

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 TLG syntax with object orientation to best serve our purpose was defined as the following: (keywords are in bold face):

class class-name
    Parameter {, Parameter} :: Data-type {; Data-type}.
    Function-name [: Function-body {; Function-body}].
end class [class-name]**.**

where Parameter $\in V_m$,
    Data-type $\in V_m \cup \Sigma \cup E$,
    Function-name $\in V_h$
    Function-body $\in V_h \cup E$[3].

The union of Data-types on the right hand side forms the type (the possible derived string) for Parameters. Multiple Function-bodies are alternative productions for Function-name, where each Function-body can be substituted by:

[2] This is why TLG has very high flexibility and natural language-like readability.

[3] Notice ($E-V_h$) will be in Function-body that is most likely be code fragments enclosed in "" in code generator.

    [Function-call {, Function-call}]

Function-call $\in V_h \cup E$. The next function-call will be called only if its previous function-call returns true, i.e. the substitution process has passed the previous term successfully. If a function-call fails somewhere, the subsequent function calls will be abandoned and the next alternative function-body should be tried if there is one.

The generative Domain Model (GDM) [Cza00] for middleware code generation in an ICB organization is formally specified by TLG. We give some examples of UMM (Figure 2) at the component level and the generator specification (Figure 3) at the system level. We ignore the component model KB at this point. On the experimental phase, our algorithm for composing heterogeneous components of client-server category will work as follows: we will generate a proxy client for a server component and a proxy server for a client component, and the service will be relayed from the original client to proxy server, then to proxy client, finally reaching the original server object. In the mean time, a bridge driver will be generated to evoke and build a common context between two proxies. For a complete explanation of this pattern, please refer to [Zha02].

*/*class CorbaAccountServer is the root class for this component where you can start to explore various aspects of this component by instantiating their respective class. Interface class tells computational (lexical, syntactical and part of semantic) aspect of this component. Class Model and Housekeeping contribute to deployment aspect. Model class states the features of the component model used in this component, and HouseKeeping class lists some packaging, compiling options and security issues. We ignore the cooperative and economic aspect in this paper. TLG keywords are in bold face. */*

class CorbaAccountServer.
   Interface :: Interface.
   Model :: Model.
   QoS :: QoS.
   HouseKeeping :: HouseKeeping.
   ServerClass :: CorbaAccountServer.
   ServerObject :: corbaAccountServer.
   ID :: maury.cis.uab.edu/CorbaAccountServer.
end class.

/ "Void", "Float" are built-in variables, they can only be substituted by their corresponding data literals. The*

declaration for "SomeExceptions" is not shown. Notions such as "deposit" belong to both E and V_h. **/*

```
class Interface.
   Void deposit Float: .
   Float withdraw Float throws SomeExceptions: .
   Float displayBalance Void: .
end class.

class Model.
   ModelName :: corba.
   ProductName :: orb2.
   OrbPackage :: "org.omg.CORBA.ORB".
   TradingServicePackage :: "com.twoab.orb2.Trading".
   HolderPackage ::
"com.twoab.orb2.TraderPackage.OffersHolder".
end class.

class QoS.
….
end class.

class HouseKeeping.
    Packages :: CorbaAccountServer .
    Imports :: .
    PolicyFiles :: "java.policy".
   CompileOptions :: "javac -classpath
%ORB2%\lib\orb2.jar; %ORB2%\lib\orb2tdr.jar" .
end class.
```

Figure. 2 Part of UMM for component CorbaAccountServer

*/*For readability, we separate comments from the code with referencing numbers. */*

```
class Generator. ----------------------------------------1
  ProxyClient :: Corba, Client. ------------------------2
  ProxyServer :: Rmi, Server.
  Mapper :: InterfaceMapper. --------------------------3
  ServerClass :: Server.SeverClass.
  ServerObject :: Server.ServerObject. ----------------4
  ClientClass :: Client.ClientClass.
  ……
  Mapper.map from ClientClass to ServerClass : . ----5
  generate ProxyClient for ServerClass return :  ------6
      "package" ServerClass.HouseKeeping.Packages ";"
       "import" ServerClass.HouseKeeping.Imports ";"
       "public class  ProxyClient { "
           "private" ServerClass ServerObject  "=null ;" ,
       ProxyClient.setupCode , ----------------------------7
       Mapper.get map from ProxyClient to ServerClass
with ServerObject , -----------------------------------------8
      "}".
 generate ProxyServer for  ClientClass return : ….
 generate  BridgeDriver for  ProxyClient and
ProxyServer return :    ….
end class
```

Figure. 3 Middleware generator class for composing RMI clients and Corba Servers

```
package CorbaAccountServer;
import ;
public class ProxyClient {
  private CorbaAccountServer corbaAccountServer=null;
  public void init() {
    initialize the ORB;
    invoke the trading service via the ORB;
    get corabaAccountServer object  using trader;  }

// The service requests are forwarded to the
corbaAccountServer.
  public void deposit(float amount){
            corbaAccountServer.deposit(amount);       }
  public float withdraw(float amount) {
        return corbaAccountServer.withdraw (amount);  }
  public float displayBalance()   {
            return corbaAccountServer.displayBalance();  }
}
```

Figure. 4 ProxyClient.java

1. Generators are specific, namely, for different component model pairs we will have different generator specifications. But this generator should be reused for the glue and wrapper code generation for all the components of the same component model pair.

2. Classes Corba and Client are predefined and are stored in the ds-KB. Those two classes act as feature models for CORBA technology domain and client domain for client-server architecture. All the classes in the knowledge base are predefined with respect to the generator.

3. InterfaceMapper is part of the KB that has rules for resolving operation mapping between two components, such as parameter wrapping.

4. ServerObject is used to relay the service from ProxyClient to CorbaAccountServer. The value can be obtained from the main program via the variable Server, where we get the value in turn from the UMM of the component.

5. Assume after this operation, we can get service mapping directly from Mapper variable.

6. This is the function that generates ProxyClient (Figure. 4). This function signature is specialized in the TLG interpreter as a built-in operator for code generation. The function body will be copied to the result file with meta variables substituted by their values.

7. Most of the setup code has already been defined in classes Corba and Client in the ds-KB. ProxyClient will automatically have those predefined features because it is defined by the product domain of both Corba and Client.

8. This method will get the operation mapping between ProxyClient and CorbaAccountServer as can be seen in Figure 4.

3. Related Work and Conclusion.

We use TLG as the formalism in this paper for two primary purposes: knowledge representation and feature modeling for various aspects of distributed and heterogeneous software components; supporting automatic middleware code generation.

TLG has very attractive properties to fit in the aim of this paper and the ultimate goal of our research thereafter.

With the natural language like syntax, a TLG specification is self-descriptive and very understandable; therefore TLG has more potential to be mastered by software engineers than other formal methods such as Z [Spi89].

XML is best for data exchange and description, but not for code generation or even more complicated task like model transformations. In a pure sense, XML carries no more semantic meaning than HTML. XML itself doesn't compute but relies on the intelligence of non-reusable XML processing engines, while TLG is Turing complete with very nice logic and functional language style reasoning. And frequent use of angle bracket templates in Xpath and XSLT [Cle01] makes the generator poorly readable.

TLG is Object-Oriented (OO) so as to be a good candidate for formal specification of OO computing entities. Nevertheless, TLG goes beyond OO programming language with its unique syntax and semantics. A simple rule such as:
NewObject:: {Object1}* Object2, Object3; Object4.
states a rather complicated feature selection and federated construction of the NewObject. It would require a big block of statements in an object-oriented programming language to show the same meaning. Plus it's very easy to combine objects and flat entity (literals) together as features since both terminals and nonterminals are allowed on the right hand side of meta-rules.

Using a formal grammar like TLG for feature modeling instead of feature diagrams [Cza00] can automate the feature configuration validation and constraint checking [Jon02], leveraging widely available open parser and type checker generator facilities such as CUP [Cup99].

The first level of context-free grammar is essentially a term-rewriting machine, whereas the second level context-free grammar together with the consistent substitution sets the context for the first one: rules and logic for applying patterns. This substitution process is very suitable for plug-and-play component composition. Hopefully, we could eventually develop TLG as a giant compiler that transforms more abstract patterns (models) to more concrete patterns (models) by steps of substitutions.

4. References:

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

[Bry02] B. R. Bryant, B.-S. Lee, "Two–Level Grammar as an Object-Oriented Requirements Specification Language," *Proc. 35th Hawaii Int. Conf. System Sciences*, 2002, http://www.hicss.hawaii.edu/HICSS_35/HICSSpaper/PDFdocuments/STDSL01.pdf.

[Bur02] C. C. Burt, B. R. Bryant, R. R. Raje, A. M. Olson, M. Auguston, "Quality of Service Issues Related to Transforming Platform Independent Models to Platform Specific Models," to appear in *Proc. EDOC 2002, 6th IEEE Int. Enterprise Distributed Object Computing Conf.*

[Cle01] J. C. Cleaveland. *Program generators with XML and JAVA.* Prentice Hall 2001.

[Cup99] Cup parser generator for Java. http://www.cs.princeton.edu/~appel/modern/java/CUP/

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

[Gre74] S. A. Greibach. "Some restrictions on W-grammars." Of Proc. *Sixth Annual ACM Symposium on Theory of Computing*, 1974, pp 256-265.

[Jon02] M. D. Jonge, J. Visser "Grammars as Feature Diagrams" *proceedings of workshop on generative programming*, April 2002. http://www.cwi.nl/events/2002/GP2002/papers/dejonge.pdf

[OMG01] Object Management Group. *Model Driven Architecture: A Technical Perspective.* Technical Report. Document #ormsc/2001-07-01. Framingham, MA: Object Management Group. July 2001.

[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*, 2001, pp. 109-119.

[Saa89] M. Saacks, J. Hassell. "Two-Level Grammar As a Technique for Formalizing Programming Schemes." Proc.*17th annual ACM conference on Computer Science: Computing trends in the 1990's*, 1989.

[Sin67] M. Sintzoff. "Existence of van Wijingaarden's Syntax for Every Recursively Enumerable Set," *Ann. Soc. Sci. Bruxelles 2* (1967), 115-118.

[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*, 2002.

[Spi89] J. M. Spivey, The Z notation: a reference manual. Prentice Hall, New York, 1989.

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

[Zha02] W. Zhao. "A Product Line Architecture for Component Model Domains". *Proc. of 12th Workshop for PhD Students in Object-Oriented Systems, held in conjunction with the 16th European Conference on Object-Oriented Programming, 2002*, http://www.softlab.ece.ntua.gr/facilities/public/AD/phdoos02/index.htm