A Product Line Architecture for Component Model Domains*

Wei Zhao[1]

Abstract
The UniFrame project proposes an innovative approach to representing knowledge of distributed and heterogeneous software components and a comprehensive architecture to seamlessly integrate them for Distributed Computing Systems (DCS) of any business domain. This proposal involves: 1) The creation of a meta-model for components and associated hierarchical setup for indicating the contracts and constraints of the components; 2) An automatic generation of glue and wrappers, based on the facts and principles in meta-model and knowledge base, for achieving interoperability; 3) Guidelines for specifying and verifying the quality of components and component complexes; 4) A formal mechanism for precisely describing the meta-model; 5) A methodology for component-based software design; 6) Validating this framework by creating proof-of-concept prototypes. The work of this paper contributes to the second point: the generative automation of middleware[2] for building the interoperability.

Keywords: UniFrame, Unified Meta-Model (UMM), Internet Component Broker (ICB), Product line architecture, Generative Programming (GP), Two-Level Grammar (TLG), middleware, glue/wrapper, Quality of Service (QoS)

1. Introduction.

"Product lines promise to become the dominating production software paradigm of the new century" [SEI02]. The goal of a product line is to provide a framework, under which the components can be interchangeable parts; to apply the software reuse systematically and strategically; to realize high productivity. During the production process of software of any business domain, we will encounter components coming from different component models, so realizing the interoperability among the heterogeneous component models is the first imperative task of achieving the final business product development.

Our research is to automate the glue and wrapper code generation required to compose the components adhering to different component models. Our methodology of pursuing the interoperability of different models is established on the idea of Generative Programming (GP) [Cza00]. GP is the key technique to automate the assembly of Commercial Off-The-Shelf (COTS) components to realize a large gain of productivity in the ultimate component market. GP focuses on software families rather than one-of-a-kind systems. Given a product order specification, highly customized end products can be generated automatically from components based on the Generative Domain Model (GDM), a model of specifying a family of systems, namely, that particular domain.

In the component technology world, "domain" has three layers of classifications: 1) Business differentiated domain, the meaning of "domain" is associated with different kinds of businesses, e.g. medical care, banking management, etc; 2) Functionality differentiated domains, based on the functionality of different parts of the software, e.g. some software is specific for database access, while some might be designated for GUI based user interaction, or some others provide pure algorithms or numerical code libraries; and 3) Technology-differentiated domains, which vary according to the various component models and technologies, e.g. some components are developed in the CORBA model while some others are in RMI or Microsoft .Net. The first definition stands at the highest level, and the third definition is the most artificial and technology driven. The most important thing we can gain from this classification is that the management and arrangement of future businesses will be organized accordingly: 1) In any business domain, there will be some organizations designated as domain specific Internet Component Developer (ds-ICD), as to the Model Driven Architecture (MDA) [OMG01] direction from Object Management Group (OMG) community, ds-ICDs are responsible for translation from platform specific models

* This research is supported by the U. S. Office of Naval Research under the award number N00014-01-1-0746.
[1] Department of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, AL 35294-1170, U.S.A., zhaow@*cis.uab.edu.*
[2] In this paper, middleware has the same meaning as glue/wrapper.

(PSMs) to executable implementations [Bur02]; 2) In any business domain, a domain specific Internet Component Exchange and Assembly (ds-ICEA) center resides at a well-known location (e.g. URL) where the users can order the product and the ds-ICDs can advertise their components, and this system assembler is the one who owns the big domain specific Knowledge-base (ds-KB) for its application domain and specialized at developing business domain model from a logic perspective; 3) There will be a group of organizations working as the Internet Component Broker (ICB), who develop and maintain the ds-KB for component models and technologies and are responsible for developing the necessary technology parameters for the mapping from platform independent models (PIMs) to PSMs; 4) Driven by the demand of high quality and reliability of COTS system, it is expected that there must be some "official" organizations for certifying and insuring the QoS provided by the individual components, their associated service price and the predicted QoS of the system after composition, e.g. some insurance company can be extended to have this role, and we might need to have an International Component QoS Authorization (ICQoSA).

Toward this ideal and promising world, the UniFrame project spans several dimensional organizations mentioned above. This paper concentrates on the third definition of the "domain". We act as an ICB for building the infrastructure for interoperability among various component model domains in DCS. To facilitate our work, we simulate the role of ds-ICEA and ds-ICD for the Banking Account Management domain.

The rest of the paper is organized as the follows. Section 2 discusses the overall picture of the system architecture. Two Level Grammar (TLG) as the formalism in our framework is briefly mentioned in section 3. Three essential activities for product line architecture for component model domains (order requirements, configuration knowledge and feature modeling representation, and production plan) are described in section 4, 5 and 6 respectively. The paper concludes with section 7.

2. System Architecture.

In the UniFrame project, we propose a unified meta-component model (UMM) to formally and uniformly represent the computational, cooperative, economic and deployment knowledge and requirements of the distributed and heterogeneous components, and a Unified Approach (UA) for integrating them [Raj01a]. 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 and what algorithms used tell the semantic view of the component. COTS components are required cooperate with each other, and the UMM cooperative aspect takes care of the interrelationship among the components such as expected collaborators. Economic aspect includes the QoS provided by this component, associated price and trading policies, and so on. Some Deployment issues such as operating system platforms, underline network quality, component model and technologies used, etc. constitute the deployment aspect of the UMM. UMM is formally represented by TLG classes. The lexical, syntactic, and semantic knowledge are encapsulated in the Interface class, and QoS is presented by a QoS Class, the features of the component model is shown in the Model class, etc. (an example is given in section 5). A creation of a software solution for a DCS using UA consists of two levels: a) the component level – developers create components, test and validate the appropriate functional and non-functional (QoS) features, register their UMM on the corresponding ds-ICEA and deploy the components on the network, and b) the system level -- a collection of components, each with a specific functionality and QoS, are obtained from the network, and an automatic generation of a software solution for a particular problem domain is achieved.

In the implementation level of UA, there are three major things need to be thoroughly understood: UniFrame Resource Discovery Service (URDS) [Sir02], ICB and ds-KB. For any business domain, there is an organizational group including one or more ds-ICEA and ds-ICD. The URDS is the infrastructure of a ds-ICEA for automated discovery and selection of components that meet the necessary functionality and QoS requirements. As the consequence of natural federation of domains and sub-domains of business definition, the natural hierarchical structure of URDS and component composition can be envisioned. From the registered

component UMM, the URDS will get a reference for individual component active registries [Sir02] (e.g. RMI registry, CORBA trading/naming service), and thus get the references for actual components inside each ds-ICD. The ICB is run by some organizations serving all the business domains for building the interoperability for their business components. ICB will be called from ds-ICEA centers along with their request for middleware facility. A ds-KB is run by this ds-ICEA to provide domain specific knowledge to all the business related company in its domain. The user interface is provided by ds-ICEA. From the client's prospect, the infrastructure of ds-ICEA is a generative library, because ds-ICEA can cache any available better solutions (in the sense of better algorithms and better QoS) provided by ds-ICD, which means the final product can dynamically evolve without changing the client provided the new solution meets the configuration knowledge and requirements specifications; at the same time, as long as the ds-ICDs provide the same functionality and QoS specified in UMM, the developer can freely change and enhance the private component implementation. The overall picture of UniFrame is shown in Figure 1.

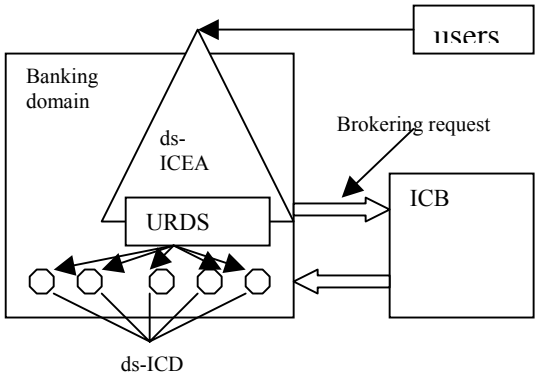

Figure 1. UniFrame Architecture

ICB is another dimensional instance of product line architecture and is the main theme of the rest of the paper. Middleware code generations for composing the components adhering to difference component models are built on a ds-KB within ICB organizations for various component models and UMM specification delivered from ds-ICEA. 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 capability to generate the glue and wrappers necessary for components implemented in diverse component models to collaborate across the Internet, [Raj01a] thus presents a collaboration vision one level above ORB.

During component assembly, QoS is an important concern that is to ensure the generated product meets required and predictable quality. The QoS requirements are expressed by selecting an appropriate set of parameters from a catalog of QoS parameters [Bra02]. QoS parameters are divided into two categories: a) static and b) dynamic. Static QoS parameters (e.g. security, parallelism constraints) are provided by the component developer and can be recognized and compared by URDS during component discovery and assembly and can be directly processed by TLG during the code generation. The QoS value of the final system is obtained by applying composition rules on static QoS parameters of individual components. [Sun02] Dynamic QoS parameters (e.g., response time) result in the instrumentation of generated target code based on event grammars [Aug97], which at run time produce the corresponding QoS dynamic metrics, to be measured and validated.

Algorithm 1: System QoS verification during the production process.

Input: a set of components to be assembled with their respective QoS parameters.

Output: The final system with optimal QoS parameters.

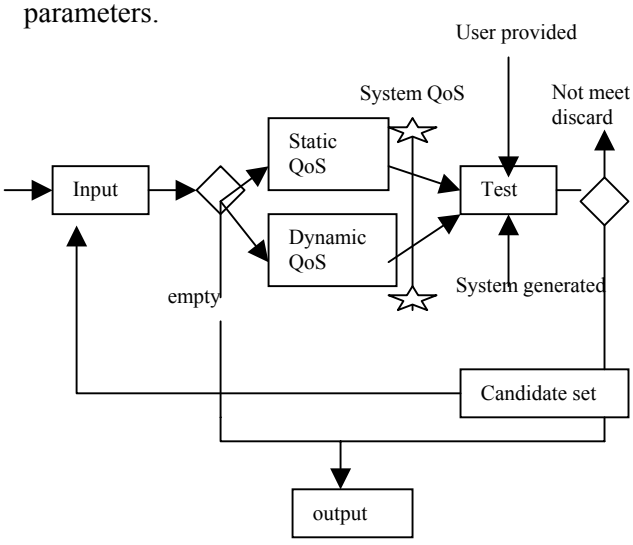

Figure 2. System QoS verification algorithm

After the completion of product generation, the user can come up with a set of test cases, or default test skeletons will be generated from the system automatically based on the QoS parameters selected by the user. If the implementation doesn't meet the desired QoS criteria, it is discarded. After that, another composition is chosen from the component collection. This process is repeated until an optimal (with respect to the QoS) implementation is found, or until the collection is exhausted. In the latter case, the process may request additional components or it may attempt to refine the query by adding more information about the desired solution from the problem domain. The QoS verification process is denoted in Figure 2. If the system assembly succeeds, a new set of UMM specifications will be generated as well so that the new product together with the new UMM will be registered with ds-ICEA for subsequent system generations.

3. Two-Level Grammar (TLG).

Two-Level Grammar [Bry02], an executable formal specification language, is chosen for internal order requirement, UMM representations, and glue/wrapper code generation. The term "two-level" comes from the fact that a set of domains may be defined using a context-free grammar, which may then be used as arguments in predicate functions defined using another context-free grammar, producing Turing equivalence [Sin67]. From the object-oriented point of view, the set of domains are a set of instance variables and the predicate functions are the methods that manipulates on the instance variables. TLG is a formal notation based upon natural language and the functional, logic, and object-oriented programming paradigms. TLG provides enough formalism for configuration rules, generation rules and domain modeling. The natural language-like characteristic of TLG makes the translation from natural language order requirements to TLG formal specification feasible and smoother. The generation rules expressed in TLG achieve the automated glue/wrapper code generation.

4. Order Requirements Specification.

The clients of UniFrame could be application system programmers or potentially be end customers. The interface between ds-ICEA and the users can be a web-based HTML form in order to facilitate the automatic ordering process. On this form, the user can specify their desired systems using Domain Specific Languages (DSL): what the problem domain and sub-domain are, what services are desired, what functionalities are needed, how good the quality of the service they expect, etc. The user can check the QoS parameters catalog provided and standardized by ds-ICEA (web-based). The query is processed using the ds-KB, such as key concepts, conventions and jargons, and use natural language processing to translate the query to TLG. The generated TLG order requirements will contain a QoS class and a functionality Interface class used to indicate the services the user expects. The application ds-KB forms part of our generative libraries and the GDM for the new system to be produced. In the sense of implementation, the query DSL is the restricted natural language (restricted by well-structured web form) refined by domain specific terms. The well-formed DSL is further a facility of the natural language processing and formalism translation, and thus benefits the matching between queries and the component UMMs.

The clients could only specify the minimum set of features of the system they want. If the specification is not sufficient to make a match, the system will provide the default setting, default dependencies and will reasonably eliminate illegal combinations. Highly customized products can be obtained by the detailed preferences such as the implementation algorithms, set of QoS parameters, or even the user's packages may be added as the part of the solution of the end system.

5. Configuration Knowledge Representation and Feature Modeling of Model Domain.

The critical part of the product line architecture for ICB is the UMM specification delivered by ds-ICEA when a brokering request is initiated. UMM is first registered at the ds-ICEA by the ds-ICD. Same as the online order system, the UMM registration is also a natural language based HTML form with the same kind of formalization translation aforementioned. UMM embeds the component configuration knowledge, feature modeling of its application, functionality and technology domains, and the generation rules for middleware needed in component assembly. After the formalization

translation, UMM will be represented by 4 TLG classes comprising the computational, cooperative, economic, and deployment aspects of components: component root class, interface class, component model feature knowledge class and component deployment housekeeping class (details are still under development). The domain feature listing along with its respective ds-KB forms the GDM for that particular domain. In ds-KBs, application domain knowledge is from the expertise of domain experts, and domain feature modeling knowledge for technology model domains is from the component model vendors. Domain feature listing in UMM for model domains will mostly be name-value pairs used to identify the entry and entry value in the model ds-KB. In Figure 3, the model domain ds-KB together with the specifications embedded in UMM form the GDM for ICB organization (denoted as the big cube in the middle). From this GDM an instance of glue/wrapper configuration can be generated automatically. With the facility of the ds-KB, the application developer (one kind of client of ds-ICEA) can possibly be freed from the component technology completely, and the ds-ICD can make the major concentration on the business functionality development and get away from the detailed interoperability and protocol mapping. We discovered that the infrastructure of the same technology domain shares the basic structure; if these common properties can be generated from the ICB as opposed to being developed by the programmer, a large amount of effort of studying new technologies and doing house-keeping programming can be saved.

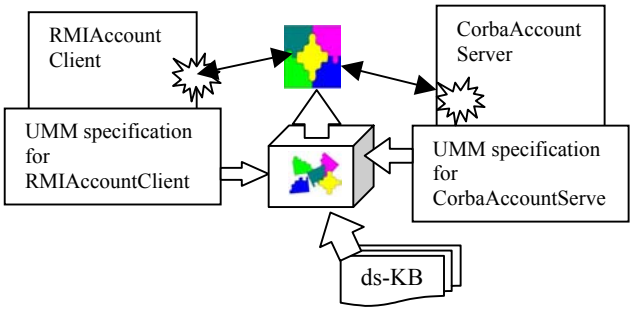

Figure 3. Product line architecture for ICB

We will use a Bank Account Management example as a brief illustration of feature modeling of model domains. Suppose, based on an order requirements for constructing a bank account management system from the user somewhere on the Internet, URDS has found three components: **RmiAccountClient, RmiAccountServer, CorbaAccountServer.** The first two adhere to the Java-RMI model and the third one is developed with CORBA technology. The UMM specifications associated with the components indicate that the two server components have the same functionality but **CorbaAccountServer** has better service guarantees and combined QoS for the final product meets that specified in the order query, thus the final system should be assembled from the RMI client and the CORBA server.

For instance, the set of UMM formal specifications generated from online UMM registration wizards for the **CorbaAccountServer** could possibly be as follows (to be completed).

*/*class CorbaAccountServer is the root class for this component where you can start to explore various aspect of this component by instantiating their respect class. We ignore the cooperative aspect in this paper. Detailed syntax for TLG can be found in [Bry02]. TLG keywords are in bold face, code fragments are underlined. */*

```
class CorbaAccountServer.
   Interface :: Interface.
   Model :: Model.
   QoS :: QoS.
   HouseKeeping :: HouseKeeping.

   ServerClass := CorbaAccountServer.
   ServerObject := corbaAccountServer.
end class.

class Interface.
   Balance :: Float.
   Void deposit Float.
   Float withdraw Float throws someExceptions.
   Float displayBalance Void.
end class.

class Model.
   ModelName :: String.
   ProductName :: String.
   OrbPackage :: String.
   TradingServicePackage :: String.
   HolderPackage :: String.

   ModelName := corba.
   ProductName := orb2.
   OrbPackage := org.omg.CORBA.ORB.
   TradingServicePackage:=com.twoab.orb2.Trading.
```

```
HolderPackage:=com.twoab.orb2.TraderPackage.Offers
Holder.
end class.

/* We will not explain QoS in this paper, details of QoS
specification may be found in [Bra02]. */

class QoS.
….
end class.

class HouseKeeping.
   PackageNames :: {String}*.
   Imports :: {String}*.
   PolicyFiles :: {Files}*.
   CompileOptions :: {String}*.

   PackageNames := orb2.Banking .
    Imports :=.
    PolicyFiles := java.policy.
   CompileOptions := javac -classpath
%ORB2%\lib\orb2.jar; %ORB2%\lib\orb2tdr.jar
end class.
```

6. Production Plan for Middleware Generation.

The production plan describes how concrete systems will be produced from the common architecture and the components [SEI02]. Our production plan of glue/wrapper code generation reflects the idea of GP. As mentioned, most of the domain features are already in the ds-KB of ICB, so is the configuration knowledge for composing these variable features. Optionally, component developers can explicitly specify their configuration knowledge and generation rules when they are asked to fill out the UMM registration wizard, but ideally, we are trying to unburden them by just requiring name and value pairs. The configuration knowledge and feature modeling are represented as TLG classes, and generation rules are shown as TLG functions. A TLG interpreter we are building right now is going to achieve the automatic middleware generation by computing the generation rule functions, because the return value we can get from those functions are regular programming language code fragments.

There could potentially be multiple configurations and combinations for components. But in practice, the ways in which the glue/wrapper code can be generated are very limited. Currently, we are working on one particular configuration for heterogeneous components belonging to the client/server category. We will continue the example stated in section 5. To compose an RMI client and CORBA server, we will have the following glue/wrapper code generated: a proxy client for a CORBA server component and a proxy server for a RMI client component (denoted in Figure 3 as two exploding stars, and their algorithmic notations are in Figures 4 and 5), and a bridge driver to glue two proxies (shown in Figure 3 as a well matched jigsaw puzzle plate, the algorithm is stated in Figure 6). Then, the proxy server redirects the service request coming from the RMI client to the proxy client; likewise, the proxy client redirects the redirected request to the CORBA server that ultimately provides the service. The two proxies are model specific access points, and they are the only entries we can get into the autonomous technology-based components. Proxies provide a common message-forwarding interface between two component models, therefore taking care of request-service mapping, data type mapping, parameters passing, etc. The bridge driver evokes and establishes the common context between two proxies and thus manages the session of the two components while the connected components are talking to each other. Upon the success of generation of the configuration for the system, this new system consisting of an RMI client and a CORBA server is assembled.

Generation rules expressed in TLG will be of the following form (it is under development). In order to make text readable, we separate the comments from the code with referencing numbers.

```
class Generator. ----------------------------------------1
  ProxyClient :: Corba, Client
  ProxyServer :: Rmi, Server ----------------------2
  Mapper :: InterfaceMapper. ----------------------3

  ServerClass : CorbaAccountServer.SeverClass.
  ServerObject : CorbaAccountServer.ServerObject.
  ClientProxyObject. ---------------------------------4
  Mapper map from RMIAccountClient to
     CorbaAccountServer.  -------------------------5

  generate ProxyClient for CorbaAccountServer:
  return ----------------------------------------------6
  CorbaAccountServer.HouseKeeping.PackageNames
     <;>
  CorbaAccountServer.HouseKeeping.Imports <;>
  <public class> ProxyClient <{>
     <private> ServerClass ServerObject  <=null ;> ;
```

```
    ProxyClient.setupCode. ------------------------7
    Mapper get map from ProxyClient to
    CorbaAccountServer with ServerObject. ------8

  generate ProxyServer for RmiAccountClient: -- 9
  ….
  generate BridgeDriver for ProxyClient and----10
    ProxyServer:
  ….
end class
```

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 should be predefined in the system to resolve the operation mapping between two components, i.e. service redirection. The definition of InterfaceMapper is not shown here.

4. ServerObject and ClientProxyObject are very important because they are used to relay the service from the RMI client to the CORBA server. The value can be obtained from the UMM of these components.

5. Syntactic mapping from service requester to service provider. Assume after this operation, we can get map domain and map range directly from Mapper variable.

6. This is the function which generates ProxyClient. Code fragements within <> are copied to the output directly. This function signature is specialized in the interpreter as build-in operator for code generation.

7. Most of the setup code has already been defined in classes Corba and Client in the ds-KB. There will be many options to change them, but not shown here. ProxyClient will automatically have those predefined features because it is defined on the product domain of both Corba and Client.

8. This method will get the operation mapping between ProxyClient and CorbaAccountServer as can been seen in Figure 4. This mapping will use the same operation signature for ProxyClient as the CorbaAccountServer because they are within the same technology model box.

```
package CorbaAccountServer;
some imports;

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

```
package RMIAccountClient;
import CorbaAccountServer package;
other imports;

class ProxyServer {
  private ProxyClient =new ProxyClient ();
  public void init() {
    register this proxy server object to the RMI registry;
    and some housekeeping handling;   }

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

Figure. 5 ProxyServer.java

```
package Bridge;
import CorbaAccountServer package;
import RMIAccountClient package;
other imports;

class BridgeDriver {
  main {
    ProxyServre proxyServer=new ProxyServer();
    ProxyServer.init();
    ProxyClient proxyClient=new ProxyClient();
    ProxyClient.init();
    Exception handling;     }  }
```

Figure .6 BridgeDriver.java

9.The hand-made ProxyServer is shown in Figure 5. In the future, it will be generated by this method. When we generate the code for ProxyServer, the Mapper will need to solve the name mapping, parameter passing, data type mapping, etc. between operations of original RMI client and CORBA server, so that the ProxyServer in a "black-box" of RMI can correctly deliver the service to the proper operation in ProxyClient in a "black-box" of CORBA. In Figure 5, we use the same signature for simplicity.

10. The code for bridgeDriver is shown in Figure 6.

7. Conclusion and future work.

Our product line architecture for ICB raises issues of software reuse. 1) Our approach of building interoperability among all the component models should be the same as long as the components are developed in client-server paradigm, so many considerations about the architecture could be reused across all the component models; 2) Our glue/wrapper code generator for a component model pair is generic for all the components developed in these models given the ds-KB for this model and UMM for this concrete component; 3) For different application domains, there are components developed in the same models, so the assets of ICBs such as ds-KBs can be reused across application domains.

Regarding our future work, more substantial work needed to build a complete set of ds-KBs for our experimental model domains: CORBA and RMI, which requires further and deeper investigation on the internal structure of these two technologies.

The task of building an interpreter for TLG is the essential contribution of achieving automatic code generation.

The construction of two GUI-based control panels for ICB and simulated Banking Account ds-ICEA are under way. The control panels are connected to the database and file system. From the control panel of the ICB, the middleware assembler can manage the ds-KB of component model domains, display UMM information of the components under current integration process, and watch the animated conversation between the components after assembly by embedding animation trigger code into the glue/wrapper files. From the control panel for Banking ds-ICEA, the business system assembler can view client's requirements, explore feature modeling of the Banking domain, manipulate the federation of compositional components by editing their icons and graphic relationship, maintain registered UMM description of the business components, communicate with the ICB via Java Message Service (JMS), and do QoS composition and verification for the final system, request the URDS of the domain for component searching.

Our current techniques for service interface mapping between two components, and matching between order requests and UMM specification of components are based on lexical and syntactical analysis on the operation signatures, especially on the operation names. We do the domain analysis, natural language processing, and naming convention processing to get best effort on mapping. In order to get high precision on the mapping and thus get high confidence of the final product, further effort on semantic analysis is in demand.

8. References.

[Aug97] M. Auguston, A. Gates, M. Lujan, "Defining a Program Behavior Model for Dynamic Analyzers," *Proc. SEKE '97, 9th Int. Conf. Software Eng. Knowledge Eng*., 1997, pp. 257-262.

[Bra02] G. J. Brahnmath, R. R. Raje, A. M. Olson, M. Auguston, B. R. Bryant, C. C. Burt, "A Quality of Service Catalog for Software Components," *Proc. Southeastern Software Engineering Conf.*, 2002.

[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.

[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.*

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

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

[Raj01a] 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.

[SEI02] Software Engineering Institute, A framework for software product line practice –version 3.0, 2002, http://www.sei.cmu.edu/plp/framework.html

[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.” to appear in *Proc. SEM 2002, 3rd Int. Workshop Software Engineering and Middleware*, 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,” to appear in *Proc. Fifth IEEE Int. Conf. Algorithms and Architectures for Parallel Processing*, 2002.