Automated Glue/Wrapper Code Generation in Integration of Distributed and Heterogeneous Software Components

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

Rajeev R. Raje, Andrew M. Olson
Computer and Information Science Department
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

UniFrame is a framework to help organizations to build interoperable distributed computing systems. Using UniFrame, a new system is built by assembling pre-developed heterogeneous and distributed software components. UniFrame solves the heterogeneity problem by explicitly modeling the domain knowledge of various technology domains (component model domains, programming language domains, operating system platform domains, etc.), from which the Interoperation Generative Domain Model (IGDM) straddling the technology domains can be constructed. The glue/wrapper code that realizes the interoperation among the distributed and heterogeneous software components can be generated from the IGDM. In this paper, an informal implementation in Java of glue/wrapper code generator is given, followed by a discussion on a formalization of IGDM. The formalism comes from the fact that if the family of glue/wrapper code can be modeled formally, an instance glue/wrapper code can be generated automatically. In this formalization, the IGDM is formally modeled as a language definition using a grammar; the code that realizes the interoperation is a valid sentence derivable from the grammar, and will be generated automatically from the IGDM during the assembly time.

1. Introduction

In today's world, distributed computing systems (DCS) are omnipresent. The successes of organizations will largely depend upon their abilities to create robust and effective software for DCS. Despite the achievements of component-based software engineering in distributed computing environments, the inherent complexity, de-centralization and heterogeneity of DCS still remain risks and challenges. Achieving a seamless interoperation among heterogeneous distributed components would be the most critical task of building a successful DCS. UniFrame [Raj01], [Raj02] is such a framework to help organizations to build interoperable DCS.

To meet the challenges, UniFrame has the following three specific goals:

1. The genetic diversity and complexity of the world (a plethora of component models, programming languages, operating systems, communication protocols) causes separation and isolation among the technology islands. UniFrame provides a unified interoperation among the collaborating components.
2. The rapid technology evolution makes the application integration a real challenge. With the interoperability, the legacy features can be integrated into the system developed in new technologies.
3. The advances in the processor and networking technologies have changed the computing paradigm from a centralized to a distributed one. "The network is the computer." The ability to deal with distribution is essential to develop large scale DCS.

In short, UniFrame aims at the distribution and interoperation. Using UniFrame, a new system is built by assembling pre-developed heterogeneous and distributed software components. This paper will discuss the interoperation framework in UniFrame.

The paper is organized as follows. Section 2 distills some aspects of UniFrame that are relevant to the

discussion of the interoperation framework. The interoperation framework is presented in section 3 with two alternative implementations (informal and formal). Some representative related work is given in section 4. The paper concludes in section 5.

2. Overview of UniFrame

Before we detail the interoperation framework, we first introduce the basics of the UniFrame.

2.1. Fundamental Theses of this Framework

Modularity and component-based software engineering. Component-based Software Engineering (CBSE) and related technologies have demonstrated their strength in recent years by increasing development productivity and parts reuse. The implementation of UniFrame is built upon the maturity of component-based software engineering [Szy02]. In our framework, features are standardized domain services. They are the smallest and the most abstract units for reuse and re-construction. One or more services are developed as a single component. Given all the possible elementary services for a business domain, a wide spectrum of systems can be generated by various combinations of services. Components are registered to the native registry in their domain for later discovery, composition and trading. Components are alive on the Internet, offering their services, QoS assurance and associated price. 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.

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 [Cle01], [SEI02], [Wei99] with the goal to automatically generate concrete software products from a domain-specification and reusable components. System family engineering has two levels: domain engineering and application engineering [Kan98]. 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. Application engineering is the process of producing concrete systems using the reusable assets developed during domain engineering. In GP, a model of a family of products is called the Generative Domain Model (GDM). The major constitutes of a GDM are a feature model for modeling the commonality and variability among the products, a generator to generate a specific product based on the feature model specification, and the implementation of reusable components from which the product can be generated. This concept of paradigm shift is the core design of UniFrame as well as the interoperation framework in UniFrame.

Capture, formalize, model and reuse engineering knowledge. Any software system has domain-specific concepts and logic, a structure, and an implementation in 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: 1) the business executives, 2) the domain experts, 3) the software managers and engineers, and 4) the software developers. During the software production process, the decisions made by all these participants contribute respectively towards: 1) the goal of the system, 2) detailed business logic of the system, 3) specifications of software architecture and developers' role assignments, and 4) concrete software development by applying different programming languages and component-based technologies.

However, when we move the development paradigm to the product-line assembly, with the goal of manufacturing the concrete software products from the GDM automatically, the engineering knowledge specific to that end product must be captured, modeled and formally defined in a domain model to guide the automated manufacturing in the application-engineering phase.

The applicability of a domain is flexible. A domain is a set of current and future applications that share a set of common capabilities and data [Kan90]. Based on the principle of separation of concerns, we have encountered different categories of domains in the process of automated product generation [Zha02]:

1. Business domain: ontology for business concepts, logic and hierarchical structure.
2. Architecture domain: ontology for software architectural patterns, software parts' functionality, role and collaborations.
3. Technology domain: ontology for implementation technologies, such as component models, programming languages, security methods, and hardware platforms.

The principle of autonomy and separation of concerns naturally shapes the categorization of those three domains. Different dimensions of engineering knowledge are built and maintained by different group of people with different education background and talent set. This gives them the opportunity to be more productive and

concentrate on the essence of their job. For example, architecture and technology domain builders are more likely to have computer science education than business domain developers.

2.2. The Structure of the UniFrame Framework

As shown in figure 1, there are two phases in UniFrame: domain-engineering and application-engineering. The domain-engineering phase simulates the domain development of three-dimensional domains (business domains [Zha04], architecture domains and technology domains). As part of the activity in business domains, designated programmers implement business domain features as software components with facilities of Model Driven Architecture (MDA) [Fra03]. Components are registered to native component model registries (e.g., RMI registry, CORBA naming services registry). Along with a natural hierarchy of business organizations, a set of available components for an application are not limited to reside on one computer, one network or one organization. They will be dispersed over the Internet. So, component searching is one of the major concerns in UniFrame. The UniFrame Resource Discovery Service (URDS) [Sir02] searches federated native component registries in the business domain for matched components. Domain level development provides the meta-data and reusable assets for the application engineering.

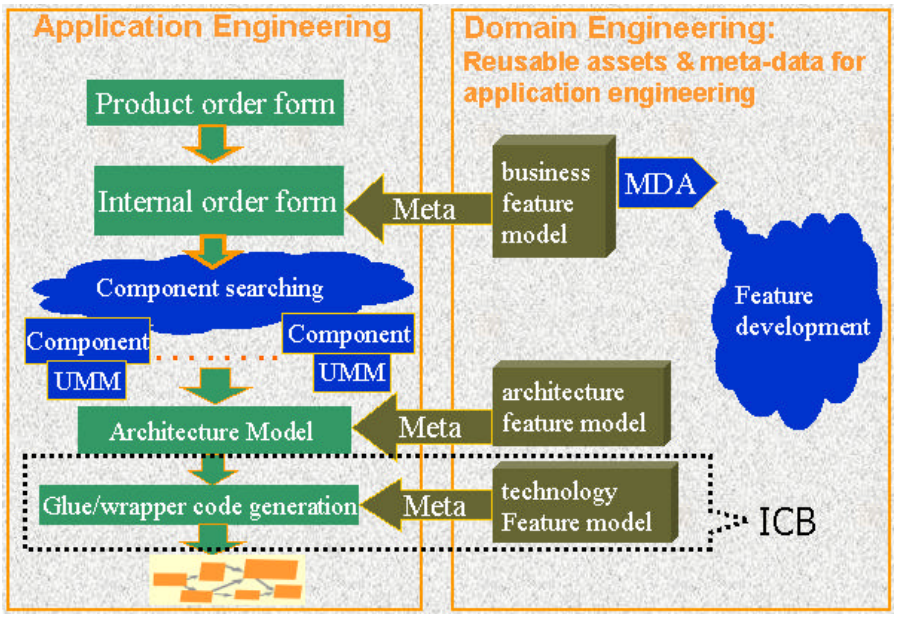

Figure 1. An Overview of the UniFrame Framework

The application-engineering phase is the process of manufacturing concrete products from the business domains. An order of a product is placed by using a user-friendly form such as HTML form, a GUI builder, a UML[1] model, a Generic Modeling Environment (GME) [GME] model or natural language. This order is translated into the internal representation that can be used for validation and initiating a search. We chose the XML for the internal representation. Then, the order is first validated according to the feature model in the business domain (no business logic violation [Zha04]). If this validation succeeds, URDS is invoked for searching the implementation components over the business domain space. When the URDS returns, a dummy composition of a set of candidate components is validated according to the feature model in the architecture domain (no architectural violation) with any necessary architectural instrumentation code generated automatically. Finally, if there are any incompatibilities in the component implementation technologies, the glue/wrapper code should be generated for the interoperation.

This paper will focus on the UniFrame interoperation framework that is called the Internet Component Broker (ICB), which is analogous to an Object Request Broker (ORB). As opposed to providing the capability to generate the glue and wrapper necessary for objects written in different programming languages to communicate transparently, the ICB provides the interoperation for

[1] Unified Modeling Language, http://www.uml.org/

components implemented in diverse component models and thus presents a collaboration vision one level above the ORB. For the interoperation of heterogeneous software components, ICB gives a vision of unified middleware.

2.3. Unified Meta-component Model (UMM)

Because of the separation of component implementation and component assembly, a unified component introspective mechanism is needed for the integration of components developed in diverse technologies. The Unified Meta-component Model (UMM) [Raj00] is such a mechanism that provides an abstraction for each component.

Our study has discovered that any individual feature implementation (component) reveals four aspects of knowledge in regards to the assembly process: computational, cooperative, deployment and economic aspects. As the domain grows, feature development would span multi-organization, multi-region/country, multi-time period, and multi-technology, which lends them a distributed and heterogeneous nature. UMM can formally and uniformly represent four aspects:

1. UMM computational aspects indicate implemented services, algorithms used, complexity, service contracts (component interface), service usage patterns. Parameters in UMM computational aspects identify features in the business domain.
2. Components are developed for reuse. UMM cooperative aspects take care of the interrelationship among the components, the individual functionality role contributing to the whole system, etc. Parameters in UMM cooperative aspects identify the entity and entity relationship in the architecture domain.
3. Some deployment issues such as component model and programming language used, operating system platforms, underlying network quality, CPU and memory usage, etc., constitute the deployment aspect of the UMM. UMM deployment aspects present the technology domain features for generating interoperation and deployment instrumentation code.
4. UMM economic aspects straddle business, architecture and technology domains, identifying the QoS parameters in each domain.

If the system assembly succeeds, a new UMM specification will be generated as well by composing component UMMs so that the new product can act as a reusable component for subsequent system generations.

There are several ways to develop UMM:

1. UMM is first documented in natural language, and then transformations can be applied to transform the informal UMM specification to formal models, and finally to the implementation software components [Bry03], [Lee02a], [Lee02b].
2. UMM is developed as a design model (e.g., UML) or a domain-specific model (e.g., GME), then a MDA approach is adopted to transform a business model to a Platform Specific Model (PSM) [Fra03], which will generate APIs, which will then be fine-tuned with concrete implementations.
3. Components are developed first, and then UMMs are generated from the implementation via some tool support.

Currently in our prototype, UMM is in a mix of natural language and XML, and can be generated from a Platform Independent Model (PIM) developed in GME [Cao03]. The components are developed manually by the programmer conforming to the feature specifications.

2.4. Quality of Service (QoS)

During component assembly, QoS is an important concern to ensure that the generated product meets the quality of service in the product order requirements. The QoS requirements are expressed by selecting an appropriate set of parameters from a catalog of QoS parameters [Bra02], [Raj02]. We have summarized and published 18 QoS parameters. QoS is business related (speed of the car, the aliveness of a supply chain), architecture related (structure integrity) or technology related (security level, turnaround time). QoS parameters are divided into two categories: a) static (the value can be obtained from UMM, such as encryption level), b) dynamic (the value can only be obtained from composition run-time, such as turnaround time). By using event grammar [Aug97], the dynamic QoS provides dynamic metrics that can be generated during the assembly time and be weaved into the glue/wrapper code. For example, we can use AspectJ[2] to weave in the turnaround time testing probe into the glue/wrapper code.

It is always possible that URDS will find multiple components with compatible static QoS, and so the dynamic QoS metrics will further refine the candidate set to generate a system that meets the user's QoS expectation of the final system.

[2]AspectJ project, *Eclipse.org*, http://www.eclipse.org/aspectj/index.html

3. Interoperation Framework in UniFrame

In this section, a detailed discussion of the interoperation framework in UniFrame is given followed by two alternative ways of implementation.

3.1. UniFrame Interoperation Framework

Potentially, there are several ways to establish the interoperation among the heterogeneous and distributed software components:

1. Source-to-source transformation: completely translate a component into the technology of its communicator. One example would be to use program transformation for legacy component migration [Bax04]. This type of technology is usually used during the reengineering [Ben87] of legacy systems. But source-to-source transformation can not be used as a general solution for the interoperation of heterogeneous software components because the complexity involved in establishing interoperation is $O(n^2)$. Considering there are n components, n(n-1)/2 transformations are needed for a full connected interoperation among n components. Despite the complexity, the source-to-source transformation is generally considered hard, and normally has to depend on a sophisticated commercial tool such as the Design Maintenance System [Bax04].
2. Transforming communicating components into a common technology for interoperation will significantly lower the interoperation complexity to O(n) since only n transformations are needed to transform n components into a common technology. An obvious example is using XML as an exchangeable technology for interoperation among different data forms.
3. Meta-interoperation is a specialization of the second item above. The common entity in meta-interoperation is not (or not only) the common technology used, but (also) is the meta-data for the transformation. Apparently, XML Meta-data Interchange (XMI) [Gro02] falls in this category, e.g., XMI defines a standard schema for object-XML mapping so that different objects can be mapped to a unified XML. MDA for the purpose of interoperation among different technologies is another example. MDA defines the standard mapping from a common Platform Independent Model (PIM) to different Platform Specific Models (PSMs) so that components in one PSM can interoperate with components in another PSM. CORBA [Vin97], [Corba] for interoperation among distributed components that are written in different programming languages also belongs to this category because the IDL can be considered as a PIM.
4. Three items listed above are all targeting translating the communicators. However, source-to-source semantic translation of software components, model (in the case of MDA), or APIs (in the case of CORBA), is laborious and error prone. The last possibility for interoperation is translating the communications instead of communicators. In terms of the size of the entity to be translated, the communication in general is magnitudes smaller than the communicators themselves. As a result, translating communication is the lightest way of establishing interoperation, which is usually realized by messaging. UniFrame has subscribed to this approach.

Before detailing the UniFrame interoperation framework, we first introduce the hypothesis we adopted. In the vision of UniFrame, components are autonomous and live in their own technology territory. In such territory, there is a central registry where components can be registered and be invoked from. Components, after being manufactured, should be registered to the registry. By autonomy, components are totally blind to any other component technologies. If a component is aware of its collaborators, it is expecting its collaborators are of the same technology as itself. Each component offers some services that are identifiable in terms of business domain features.

Thus, the interoperation means the communication across the territory boundaries. There are two main tasks in this communication: first, where is the component; second, how do components communicate. URDS [Sir02] takes care of the first task by searching federated registries in the business domain for expected components and returning with the registry and the component ID. This paper specifically addresses the second task. The interoperation is achieved by generating proxies dynamically for invoking the components from the registry and for replaying communications. Shown in the figure 2, the communication between the component and the proxy falls in the same territory. The essential aspect of interoperation in this picture is to establish a common message protocol so that proxies can talk to each other across the technology boundaries. In UniFrame, we use Simple Object Access Protocol (SOAP)[3] for encoding and decoding parameters, data types and exceptions. The code that actually realizes the interoperation is called the glue/wrapper code, which includes two proxies.

[3] SOAP Messaging Framework, W3C, http://www.w3.org/TR/2003/REC-soap12-part1-20030624/

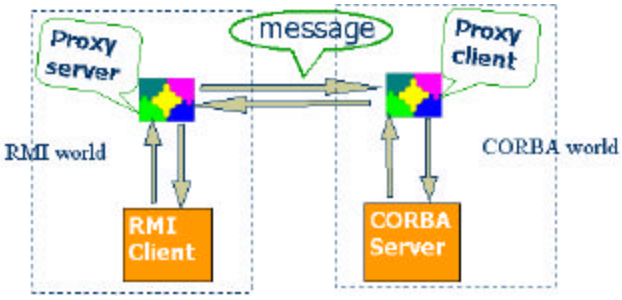

Figure 2. The Interoperation Framework

To be specific, the fundamentals of the UniFrame interoperation framework are as follows:

1. The glue/wrapper establishes a binary connection for any two heterogeneous components. Between these two components, one must be the service requester, and the other one must be the service provider. From this perspective, no matter what is the underlying architecture of the whole distributed system, client-server is a general framework for a binary relationship of a pair of communicating components. For the communication between the two components we need a proxy server for the service requester, and a proxy client for the service provider. The proxy server registers itself to its component registry listening for the request coming from the service requester, and then translates this request through the SOAP channel to the proxy client who decodes the SOAP message and invokes the ultimate service provider with the redirected service request. Two proxies also take the responsibility of managing the communication session. The use of proxies attacks the problem of the heterogeneity of component models; and SOAP/HTTP solves the language heterogeneity and distribution.
2. The glue/wrapper code realizes the interoperation at run time, i.e., the existing component should not be modified or recompiled. The glue/wrapper can be generated, compiled and bound dynamically during the composition run time.
3. Because the semantics of business domain features are standardized and shared by all the feature implementation developers, each implementation can have slightly customized interfaces including different naming strategies of parameters and methods, and the variations on the parameters (only to a degree that the translations can be done automatically for solving the variations).

The main challenge in realizing interoperation among heterogeneous components is not the issue of constructing glue/wrapper code for a particular pair of components, but to construct a generator that can automatically generate glue/wrapper code for different pairs of components on demand. To achieve that, the generator needs to access both the knowledge for the technology domains at the domain level and the knowledge for a particular component implementation at the component level. For the technology domain, the generator has to know how many kinds of technology domains (component model domains, programming language domains, operating system platform domains, etc.) and what information in a particular technology domain (e.g. Java programming language domain) for the interoperation purpose. At the component level, the generator needs to know from the deployment aspect of UMM what technologies are employed in a component implementation.

In the next section, we will review an informal implementation of the glue/wrapper code generator.

3.2. An Informal Implementation of Glue/Wrapper Code Generator

In this informal implementation, both the generator and the technology domain knowledge are written in Java. Domain knowledge is embedded in the java classes in the form of printing statements. Shown in figure 3, there are 4 different kinds of technology domains that the generator directly accesses: proxy client, proxy server, programming languages, and operating systems. The proxy server and the proxy client inherit the architecture knowledge from the architecture domain server and client respectively. There can be federated hierarchies in each technology domain. For example, for a specific component model, say Remote Method Invocation (RMI)[4], there is an RMIServer that implements ProxyServer and extends RMI, also there is an RMIClient (although not shown in figure 3) that implements ProxyClient and extends RMI. Then we will be able to generate both proxy server and proxy client for a RMI component. A component model is usually abstract and should be concretized by different vendor-specific technologies. For example, TAO [Har98] is a concretization of CORBA, and JavaRMI is a concretization of RMI.

There are some benefits in developing the generator in Java.

1. By taking advantages of polymorphism, the generator is generic to any specific technology as it only deals with interfaces.
2. By using Java reflection, we can dynamically load a specific technology domain class as needed based on

[4] Java Remote Method Invocation (Java RMI), http://java.sun.com/products/jdk/rmi/

the parameters in the component UMM. For example, if the UMM indicates the language used for two components are Java and C++, then only the Java and C++ classes in programming language domains are loaded into the Java runtime environment. This will drastically improve the performance of the generator considering technology domains contain a wide variety of classes.

3. The generator framework is extensible. We can extend the framework with any programming languages, operating systems and component models. In the case of new technologies (a new component model, a new vendor-specific product for an existing component model, etc.), we only need to modify the framework by adding the new technology domain subclass, and the generator should remain unchanged.

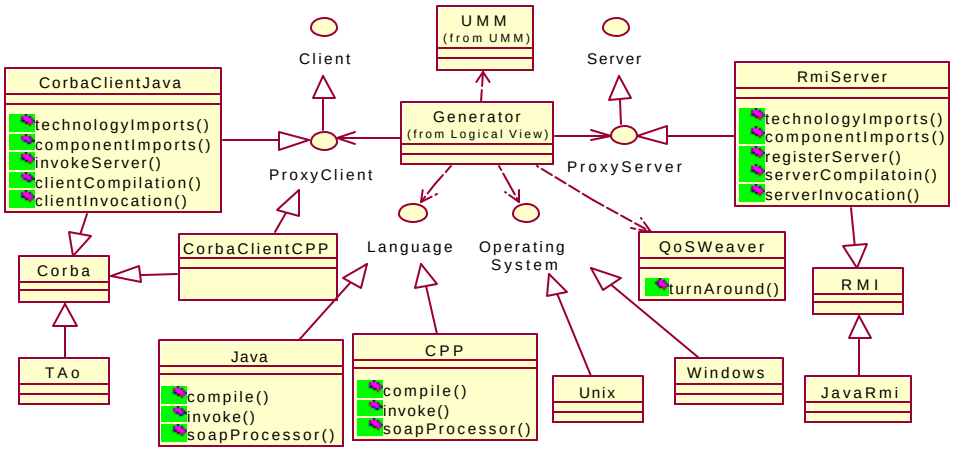

Figure. 3 The Glue/Wrapper Code Generator in Java

By constructing a technology domain knowledge base, we do not mean constructing a complete specification for a particular technology. For the interoperation under the hypothesis mentioned earlier, only some specific information is needed. Such information includes: how to register and invoke a server from a registry in a specific component model; how to process SOAP messages in a specific language; how to compile and invoke a program in a specific programming language and operating system platform; what are the component model product specific class path and compilation options.

Besides generating interoperation code, the generator has other responsibilities such as dynamic QoS testing, system monitoring, and session management probe generation. As an example, we can use AspectJ [Kic97] to weave turnaround time testing code into the generated proxies (shown in figure 4 as QoSWeaver class).

3.3. Towards the Formalization of Automated Glue/Wrapper Code Generation

In the previous section, we have sketched out some benefits of implementation in Java. However, embedding the technology domain knowledge into a programming language using printing statements tends to blur the technology domain specific information. Consequently, it will be an obstacle for domain evolution and reuse, and further prevent the generator from evolving.

To solve this problem, we have applied the Generative Programming (GP) [Cza00] and Product-line Architecture [Cle01], [SEI02], [Wei99]. Both of these technologies aim at defining and modeling a family of products so that a product instance can be generated automatically from this family. As mentioned in section 2.1, the system family development is the core design of UniFrame, and as well as the interoperation framework in UniFrame. The rationale for the applicability of GP is that the glue/wrapper code for a pair of components of particular technologies is one product instance; the glue/wrapper code for the pairs of components of all possible technologies form a family of glue/wrapper code. If this family can be well modeled, one particular glue/wrapper code instance can be generated from the family automatically.

The GDM for the family of glue/wrapper code is called the Interoperation GDM (IGDM, see figure 4). IGDM straddles different technology domains including different component model domains, different programming language domains, different operating system domains, and different security method domains. The feature model in the IGDM explicitly models the domain-specific features of different technology

domains, which direct the variations among glue/wrapper code instances. The generation of glue/wrapper code for components in different technologies depends on the domain-specific features of technology domains. In the IGDM, the reusable components, from which the glue/wrapper code can be generated, are the code fragments of potential glue/wrapper code.

In order to support the automated glue/wrapper code generation from the IGDM, we have adopted a formal modeling theory on feature modeling in the IGDM. The feature model in the IGDM is defined as a language; the glue/wrapper code generated from the IGDM is a valid sentence of this language. The generator for the glue/wrapper code is the interpreter for the grammar that is used to define the feature model. The terminal symbols of the grammar are code fragments. The glue/wrapper code is a string of code fragments.

To apply successfully this theory and the programming-language-oriented techniques to feature modeling, the first question to be answered is whether there exist concepts in feature models that are the counterparts of syntax and semantics in programming languages. The fact is these concepts do exist in the feature models, and are discussed below.

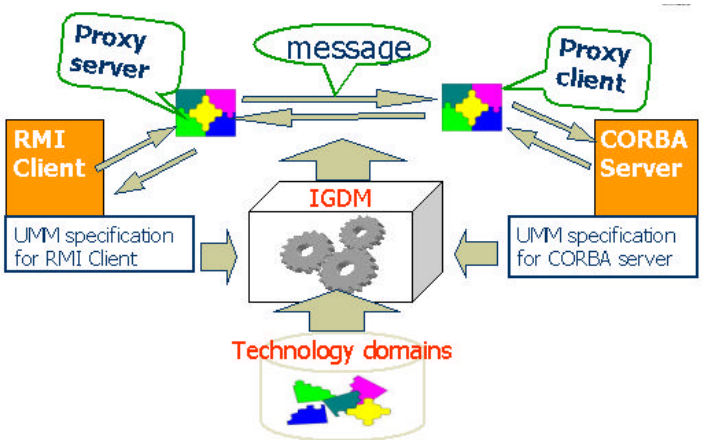

Figure. 4 The Formalization of Automated Glue/Wrapper Code Generation

1. The composition syntax is the structure of the interoperable framework. The following context-free derivations show part of the structure of the glue/wrapper code to be generated. Currently, the grammar we use to define IGDM feature model is called TLG++ [Zha04]. The following code is in TLG++. For the notational syntax, the "," is for "and", and the ";" is for "or".

```
glueWrapper code : proxyServer,
  proxyClient.
proxyClient : technologyImports,
  componentImports, invokeServer,
  clientCompilation, clientInvocation.
invokeServer : findRegistry,
  getServerObject, initiateServer,
  serverInvocationExceptions.
……
```

2. Static semantics constrains types of glue/wrapper code to be generated. In particular, the component model is modeled as the type of the component; and programming languages, operating systems, message's signature and type, security methods, and digital signatures are modeled as the attributes of the components. Based on the different value of component type and its parameters, different glue/wrapper should be generated. In the following code fragments, the codes in bold are the parameters that indicate the different features of technology domains. TLG++ distinguishes itself from context-fee grammars is this feature of parameterization. The parameters are evaluated while the syntax tree is built. The codes underlined are the glue/wrapper code fragments enclosed in the double quotation mark. The code fragments are the terminals of the grammar.

```
findRegistry:
  where ComponentModel= corba,
  "orb=  org.omg.CORBA.ORB  .init(args,
null);"
   ProductTraderPackage
  "trading=TradingHelper.narrow
```

```
  (orb.resolve_initial_references("LCBT
rading" ));"……;
where ComponentModel = rmi,
    ……;
where ComponentModel = j2ee……
```

3. Dynamic semantics models the component composition QoS that are affected by the component technologies. If the components are implemented in different technologies, they will present different QoS values. The generated glue/wrapper code will also affect the QoS, and should be part of dynamic semantics. Event grammars [Aug97] are used to generate an event trace, which acts as the QoS metric to be inserted into the generated glue/wrapper code.

4. Related Work

There have been some attempts towards achieving interoperability among different technologies emerging out of industry and research organizations. Some prominent examples, besides the work mentioned in section 3.1, are described below.

Middleware technologies such as CORBA [Corba] and DCOM [Ses97] provide a communication infrastructure for a heterogeneous and distributed collection of objects. Based on this infrastructure, objects can interoperate across networks regardless of the language in which they are written or the platform on which they are deployed. However such middleware or component models exclude the presence of others. UniFrame gives a vision of unified middleware providing the interoperation not only among the programming languages and platforms but also among the component models. The proxies in this paper are similar to the stubs/skeletons in CORBA. However, the concept of IDL in CORBA is elevated to the business feature model of this paper. The feature model in a business domain defines the semantics of features and their interactions, and is shared by the feature implementation developers.

Some ad hoc approaches for interoperation between component models come out from the industry that are targeting specific component model pairs. RMI is a language centric approach using JRMP (Java Remote Method Protocol) for interactions between distributed objects. RMI requires that the entire distributed application be programmed in pure Java. Sun[5] and IBM[6] have jointly developed RMI-IIOP, a new version of RMI that runs over IIOP and interoperates with CORBA ORBs and CORBA objects programmed in other languages. To bridge CORBA and DCOM, the Object Management Group (OMG) provides the interworking architecture specifications regarding the mappings between DCOM and CORBA which includes: Interface Mapping, Interface Composition Mapping and Identity Mapping, etc. [Rap01].

Web services [New02] claims to be a means of interoperation among component models. Nevertheless, web services achieve the interoperation by introducing yet more standards such as Web Service Definition Language (WSDL), Universal Description, Discovery, and Integration (UDDI), and SOAP. This does not completely solve the problem due to the inherited local autonomy and the difficulty of the adoption of standards, whereas UniFrame approaches the problem in a different way by modeling existing technology domains.

As mentioned in section 3.1, MDA [Fra03] has subscribed to the meta-interoperation approach. For example, for the interoperation between the web service and Java, the system has to know the following three things: the platform-independent UML class model, the UML-java mapping, the UML-SOAP/WSDL mapping. As with web services, MDA forces UML or MOF to be the standards for the interoperation.

5. Conclusions

In this paper, we have discussed an interoperation framework for integration of heterogeneous and distributed software components. The target goal of this framework is the automated glue/wrapper code generation during the component assembly time. This framework incorporates the following key concepts: 1) an introspective meta model (UMM) for the autonomous components; 2) an explicit modeling of domain knowledge of various technology domains instead of introducing new standards for interoperation; 3) introducing the IGDM that models a family of glue/wrapper code to provide a formal foundation for automated glue/wrapper code generation; 4) a language-oriented way to formalize the IGDM so that the glue/wrapper code generated from IGDM is a valid sentence that can be generated from a grammar. The initial experiments have been carried out to integrate components written in RMI and CORBA, and the glue/wrapper code can be automatically generated for their

[5] Sun Microsystems, Java RMI-IIOP Documentation url: *http://java.sun.com/j2se/1.3/docs/guide/rmi-iiop/index.html*

[6] IBM developer Works, Java technology Standards RMI-IIOP, url: *http://www-106.ibm.com/develperworks/java/rmi-iiop/summary.html*

interoperation based on the informal implementation approach. Future work will be to design and extend our grammar notation to formalize IGDM. Experiments are also done on applying this framework to other component models such as .Net, DCOM, J2EE, Web Services, mobile agents, and as well as wireless component models [Sha03].

6. Acknowledgement

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

7. 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.*, pp. 257-262, 1997.

[Bax04] I. Baxter, C. Pidgeon, M. Mehlich, "DMS: Program Transformations for Practical Scalable Software Evolution", to appear in the Proc. of 2004 International Conference on Software Engineering (ICSE), 2004.

[Ben87] S. Bendifallah and W. Scacci, "Understanding Software Maintenance Work", *IEEE Transactions on Software Engineering*, Vol. 13 , No. 3, 1987.

[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.*, pp. 513-521, 2002.

[Bry03] B. Bryant, B-S. Lee, F. Cao, W. Zhao, C. Burt, J. Gray, R. Raje, A. Olson, M. Auguston, "From Natural Language Requirements to Executable Models of Software Components", *Proc. of* the *Monterey Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation*, pp. 51-58, 2003.

[Cao03] F. Cao, Z. Huang, B. Bryant, C. Burt, R. Raje, A. Olson, M. Auguston. ``Automating Feature-Oriented Domain Analysis," Proc. of the 2003 International Conference on Software Engineering Research and Practice (SERP'03), CSREA Press, pp. 944-949, 2003.

[Cle01] P. Clements, L. Northrop, *Software Product Lines: Practice and Patterns*, Addison-Wesley, 2001.

[Corba] Common Object Request Broker Architecture (CORBA), http://www.corba.org/

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

[Gro02] T. Grose, G. Doney, S. Brodsky, *Mastering XMI*, John Wiley & Sons, Inc., 2002.

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

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

[Har98] T. Harrison, D. Levine, D. Schmidt, "The Design and Performance of a Real-time CORBA Event Service", *Computer Communications*, Vol. 21, No. 4, 1998

[Kan90] K. C. Kang, S, G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study", Technical Report, CMU/SEI-90-TR-21, 1990.

[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.

[Kic97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J. M. Loingtier, J. Irwin, "Aspect-Oriented Programming", *Proc. of European Conference for Object-Oriented Programming (ECOOP)*, pp. 220-242, Springer-Verlag, 1997.

[Lee02a] B.-S. Lee, B. R. Bryant, "Automated Conversion from Requirements Documentation to an Object-Oriented Formal Specification Language", *Proc. of ACM Symposium on Applied Computing (SAC)*, pp. 932-936, 2002.

[Lee02b] Lee, B.-S. and Bryant, B. R., "Automation of Software System Development Using Natural Language Processing and Two-Level Grammar," *Proc. 2002 Monterey Workshop Radical Innovations of Software and Systems Engineering in the Future*, 2002, pp. 244-257.

[New02] E. Newcomer, *Understanding Web Services: XML, WSDL, SOAP, and UDDI*, Addison-Wesley, 2002.

[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. 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,* Vol. 14, No. 12, pp. 1009-1034, 2002.

[Rap01] Raptis, K., Spinellis, D., Katsikas, S. "Multi-Technology Distributed Objects and their Integration," *Computer Standards & Interfaces, Vol. 23, 157-168, 2001*.

[Sha03] P. V. Shah, B. R. Bryant, C. C. Burt, R. R. Raje, A. M. Olson, M. Auguston, "Interoperability between Mobile Distributed Components using the UniFrame Approach," Proc. of the 41st Annual ACM Southeast Conference, pp. 30-35, 2003.

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

[Ses97] R. Sessions, *COM and DCOM: Microsoft's Vision for Distributed Objects*, New York, NY: John Wiley & Sons, 1997.

[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 LNCS, Vol. 2596, pp. 20-35, 2002.

[Szy02] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd edition, Addison-Wesley Longman, 2002.

[Vin97] S. Vinoski, "CORBA: Integration Diverse Applications Within Distributed Heterogeneous Environments", *IEEE Communications*, Vol. 14, No. 2, 1997.

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

[Zha02] 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

[Zha04] W. Zhao, B. R. Bryant, R. R. Raje, M. Auguston, C. C. Burt, A. M. Olson, "Grammatically Interpreting Feature Compositions", to appear in the proceedings of the 16th International Conference on Software Engineering and Knowledge Engineering (SEKE'04), 2004.