



A Component Assembly Approach Based On Aspect-Oriented Generative Domain Modeling

Fei Cao, Barrett R. Bryant, Carol C. Burt¹

*Department of Computer and Information Sciences
University of Alabama at Birmingham
Birmingham, AL, USA*

Rajeev R. Raje, Andrew M. Olson²

*Department of Computer and Information Science
Indiana University Purdue University at Indianapolis
Indianapolis, IN, USA*

Mikhail Auguston³

*Computer Science Department
Naval Postgraduate School
Monterey, CA, USA*

Abstract

We present an approach towards automatic component assembly based on aspect-oriented generative domain modeling. It involves the lifecycle covering the component specification generation, and subsequent assembly of implementation components to produce the final software system. Aspect-oriented techniques are applied to capture the crosscutting concerns that emerge during the assembly process. Subsequently, those concerns are woven to generate glue/wrapper code for assembling heterogeneous components to construct a single integrated system.

Keywords: Component Assembly, Generative Programming, Generative Domain Model, Component Specification, Aspect Orientation, UniFrame, Two-Level Grammar.

¹ Email: {caof, bryant, cburt}@cis.uab.edu

² Email: {rraje, aolson}@cs.iupui.edu

³ Email: auguston@cs.nps.navy.mil

1 Introduction

As software component development technology becomes more mature, the notion of developing software systems by assembling Commercial-Off-The-Shelf (COTS) components (implemented in models such as COM⁴, DCOM⁵, EJB⁶, CCM⁷) becomes not only theoretically rational, but also practically sound. Component-Based Software Composition offers a development paradigm with reduced time-to-market and cost while achieving enhanced productivity, quality and maintainability [3].

But component assembly remains mainly either a handcrafting effort or proprietary approach [28]. It is becoming an even harder problem when components are delivered in binary form which may need binary code adaptation [17], or when the underlying implementation language, deployment environment are heterogeneous. For the latter case, what's commonly seen is a middleware approach such as CORBA, that allows the components to work cooperatively across language and platform boundaries. However, this approach may also add extra complexity that makes the construction of a distributed system more difficult [9].

UniFrame⁸ is a framework for seamless interoperation of heterogeneous distributed components. It aims to automate the process of integrating heterogeneous components to create distributed systems that conform to quality requirements. By automatic generation of glue/wrapper code based on the developer's functional and non-functional specification ([2], [26]), the system generated will be tailored to specific requirements as opposed to being a monolithic end product, and reliability is also enhanced. In this paper, we present an approach to support the automatic component assembly in UniFrame by applying aspect-oriented generative domain modeling. In Section 2, we introduce the background information of UniFrame. Section 3 and 4 present our approach of using aspect-oriented generative domain modeling for component assembly. Section 5 presents some discussion, followed by the description of related work together with the conclusion in Section 6.

⁴ Component Object Model, <http://www.microsoft.com/com>

⁵ Distributed Component Object Model, <http://www.microsoft.com/com/tech/dcom.asp>

⁶ Enterprise Java Beans, <http://java.sun.com/products/ejb>

⁷ CORBA[®] (Common Object Request Broker Architecture, <http://www.omg.org/corba>)
Component Model, <http://www.omg.org/cgi-bin/doc?orbos/99-07-01>

⁸ Unified Framework for Seamless Integration of Heterogeneous Distributed Software Components - <http://www.cs.iupui.edu/uniFrame>

2 Background

2.1 Generative Programming

As is introduced in [10], *Generative Programming (GP)* is a software engineering paradigm based on modeling software families such that, given a particular requirement specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge. The requirement specification is sometimes referred to as *ordering* of products; the terminology used to specify family members is referred to as the *problem space*; the implementation components with their possible configurations form the *solution space*. The problem space and solution space, together with the associated configuration knowledge, constitute the *Generative Domain Model (GDM)* [10]. The distinct property of GP is that it is not only about a development *for reuse* in terms of building a GDM for software system families, but also about a development *with reuse* in terms of using GDM to generate concrete systems; it focuses on generation of system families rather than a one-of-a-kind system.

2.2 UniFrame

With advances in network technology, software systems are shifting from a closed, centralized architecture to being open and distributed; from being homogeneous in implementation to adopting heterogeneous components for constructing the whole system. To harness the omnipresent components in a distributed system while having to address the inherent complexity of such a paradigm, the functional and non-functional properties of components must be formally captured, and there needs a means to assure the specified QoS (Quality of Service)⁹ for the system assembled from components. UniFrame is a framework to address those concerns [26]. It uses a Unified Meta-component Model (UMM) [25] to encode the meta-information of a component such as functional properties, implementation technologies, and cooperative attributes.

In UniFrame, a GDM is also used to capture the domain knowledge and to elicit assembly rules. But the use of a GDM doesn't include the implementation components: this part is assumed to be offered in a distributed system environment by different vendors observing the stipulated specifications in the problem space of the GDM; those implementation components are exposed by

⁹ In this paper, “non-functional aspect”, “non-functional-property” and “Quality of Service (QoS)” may be used interchangeably.

vendors and are subject to location by a distributed resource discovery service [27]. In addition, the GDM in UniFrame is used to capture the assembly rules for the discovered components.

Figure 1 illustrates the big picture of UniFrame. The annotated number represents the processing order. Starting from domain experts, a GDM will be created (1.1) and will be used together with some domain standards (1.2) as guidelines (2.1, 2.2) for component developers to implement components in solution space. Those implementation components, after being quantified with some QoS parameters (3), will be exposed to a distributed resource discovery service (5). Thereafter, a system integrator will query into the problem space of the GDM for available/deployed component information (6), and then command the resource discovery service (7) to fetch the required components (5,8) for assembly. The component assembly is subject to validation (9) based on specified QoS requirements. If it is not validated (11), then the integrator has to initiate the query and integration process iteratively. As it can be seen from above, the GDM stands as a crucial part of UniFrame, and how GDM is represented so as to facilitate the component assembly is of vital importance. We call the means to represent GDM *generative domain modeling*, which is further detailed in the next section.

3 Overview of the Approach

3.1 Specification of Components in the Solution Space of the UniFrame GDM

Components in UniFrame are specified using the formalism of Two-Level Grammar (TLG) [4]. The specification in TLG provides flexibility in translating TLG specifications to other representations, such as other formal specification languages like the Vienna Development Method [21], or application code [6]. TLG contains two context-free grammars, one describing type domains and the other describing rules and operations on those domains. Note it is not required to have both levels. Below is a template TLG specification.

```

class Identifier-1
  Identifier-1, Identifier-m1 :: DataType1; DataType2;...;
  DataType-n1.
  Function-signature-1,...Function-signature-m2 :
    function-call-1,function-call-2,..., function-call-n2.
end class Identifier-1.

```

The line containing “::” denotes the first-level type domain definition, for which the right hand side of “::” provides the type (which is called a *meta-type*)

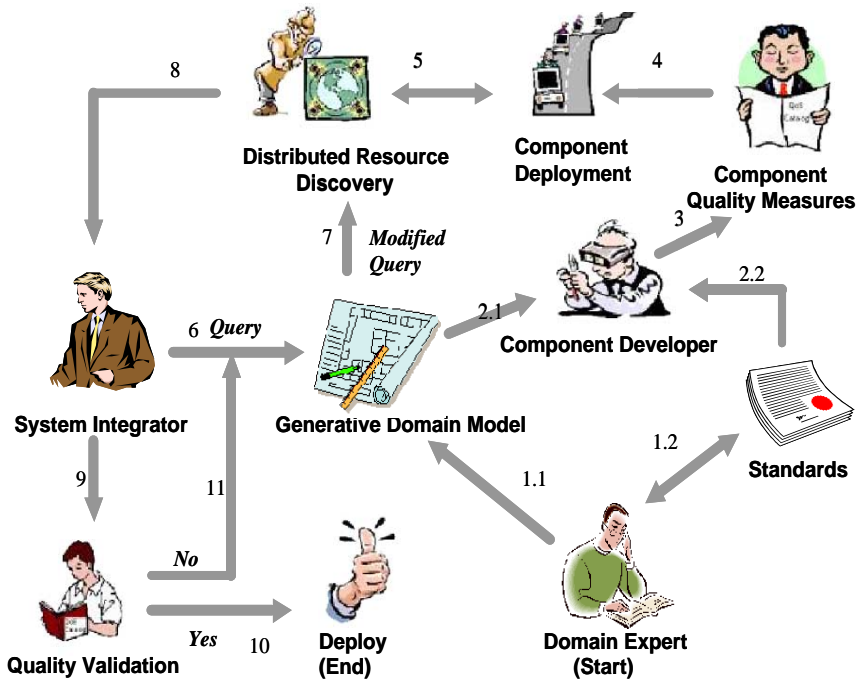


Fig. 1. the Process of UniFrame.

while the left hand side provides the variable name. Note the right hand side may specify multiple types at the same time, which are delimited by “;”. The left hand side may also have multiple variables separated by “,”, which are of the same meta-type as defined on the right hand side. Also note the meta-type may form a hierarchy (*meta-type hierarchy*). For example, *BankOperation* may be the meta-type of the *Withdraw* operation, while *Service* may be the meta-type of the *BankOperation*. Consequently, *Service* is also regarded as the meta-type of *Withdraw*.

The line containing “:” denotes the definition of the second-level rule/operation (also called *hyper-rule*) over the first-level type domains. “;” can be used in the right hand side of “:” to delimit multiple rules which share the same function signature on the left hand side. Note both first-level and second-level may contain multiple (including zero) sentences as opposed to just one sentence of each in the above description.

3.2 Separation of Concerns in Generative Domain Modeling

Consider the following two component specifications in the GDM problem space (note this simple example serves for the motivation purpose only—full

definition of a component description language is provided in Section 4.2).

```

Component BankServer
  provides AccountManagement :
    applies AccessControl
end Component

```

```

Component BankClient
  requires AccountManagement :
    uses RMIServer applying QoSMonitor
end Component

```

In the *BankServer* specification, the provided service *AccountManagement* uses *AccessControl*. But as business rules are subject to change, the *BankServer* may lift the *AccessControl* or enforce other type of controls, either of which will reduce the reusability of the original *BankServer* implementation component. In the *BankClient* specification, the “*RMIServer*” and “*QoSMonitor*” that are required for a server-side *AccountManagement* service represent the glue/wrapping logic needed to integrate the client and server components. This tangles the *BankClient* component and also reduces its reusability as glue/wrapping requirements change.

Aspect-Oriented Programming (AOP) [18] provides a means to capture crosscutting aspects in a modular way with new language constructs, and also provides a *join point model* to “hook” the aspects with the base program. This is the basis of augmenting the component specification approach with aspect orientation in order to separate those crosscutting assembly-related aspects of components. Those aspects do not need to be implemented by vendors. The separation will refine the granularity of GDM, and contribute to the *maximal combination, minimal redundancy, and maximum reuse*, which are the desired properties of implementation components [10] in the solution space of GDM. Consequently, the component assembly process evolves into an aspect weaving process. Table 1 provides the tentative catalog of assembly related concerns.

Figure 2 illustrates the aforementioned idea. The arrow ending with a diamond figure represents the *include* relationship as in the standard UML¹⁰ notation. Separation of concerns [23] is introduced into the domain analysis phase, the output of which is the GDM. The GDM includes the concerns identified at the domain analysis phase (which are also called *early aspects*¹¹), and those aspects are collectively stored into a repository called the *aspect*

¹⁰ Unified Modeling Language, <http://www.omg.org/uml>

¹¹ <http://early-aspects.net/>

Functional	Business rule enforcement
	Specific technology instrumentation
	Pre/post condition
	...
Non-Functional	Profiling
	QoS Validation
	QoS Instrumentation
	...

Table 1
Assembly Related Aspects

library. This aspect library corresponds to the configuration knowledge as indicated in Section 2.1. The GDM also includes Component Description Language (CDL, the actual definition to be provided in Section 4.2) in its problem space part; the CDL is also used as a guideline for implementation of components by different vendors. Upon an *ordering* request over the GDM problem space, the CDL in the problem space will be weaved with involved assembly aspects into the specifications for glue/wrapper code generation, which by referencing the implementation components, will be used to generate final glue/wrapper code to connect the components.

4 Multi-Stage Component Assembly

Before we detail the component assembly process in Section 4.3, we provide the related specification definitions in Section 4.1 and 4.2.

4.1 Definition and Use of Aspect

In such AOP languages as AspectJ [19], aspects are defined in a way that is closely bound to the base program (the *join point* is specified syntactically based on the base program). In contrast, in Figure 2, aspects are separately stored as a library. Thus, a *join point model* is required to hook the aspects to the targeted program so as to apply the related *advice* provided in the aspect.

Aspect Description Language (ADL) is defined as follows:

```
aspect <aspectname>
  advises: <Meta-type>.
```

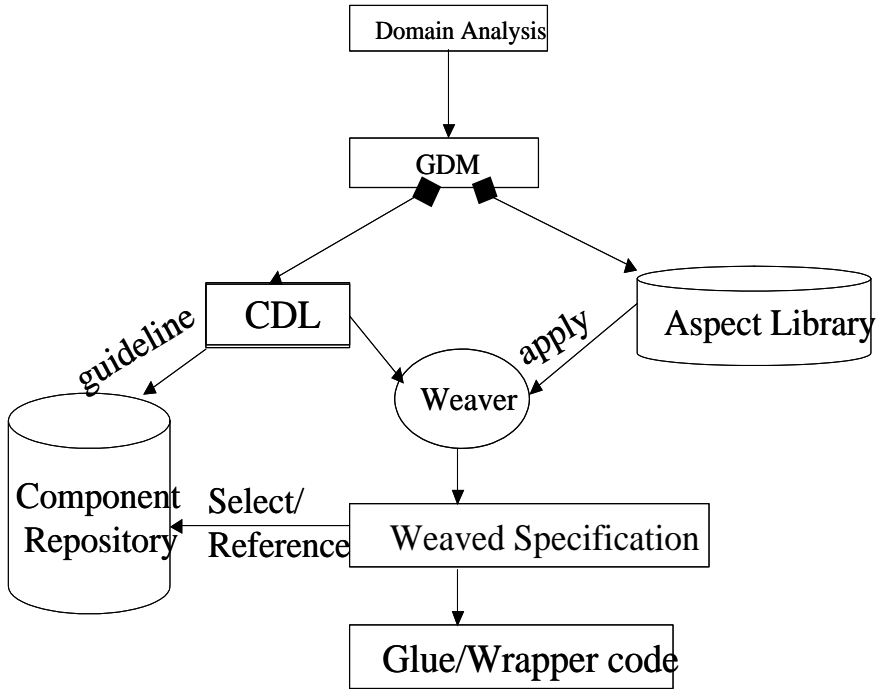


Fig. 2. Aspect-Oriented Generative Domain Modeling

```

[before: <advice>.]
[after: <advice>.]
end aspect <aspectname>

```

The name enclosed in “<>” represents a grammar variable, which will be exemplified in Section 4.3. The “[]” is used to delimit a part that is optional. Those notations apply to the following Aspect Usage Language (AUL) and CDL as well. The <Meta-type>, which is defined as in Section 3.1, is used to specify the types of domain services that this aspect can be applied. The *advice* following the directive *before/after* provides the pre/post actions to be performed or pre/post conditions to be enforced before/after the domain services, which can be used for temporal dependency specification and tracing/QoS code instrumentation. For example, in [30], before/after advice is used to specify rules for model checking. Consequently, the aspect library represents a collection of assembly rules.

AUL is defined as follows:

```

apply <aspectname> on <type> [when <relational-expression>]

```

<aspectname> corresponds to an assembly-related aspect, which already provides a means to specify assembly rules as described in the preceding paragraph. The <type> has to be consistent with the applicable <metatype> in the ADL of <aspectname>. By *consistent* we mean the <metatype> as in the ADL of <aspectname> should reside at the root position of some meta-type hierarchy (see Section 3.1 for definition), where <type> is part of the hierarchy. The *when* directive in AUL further specifies the scenarios using relational expressions, under which this aspect can be applied; in addition to the base-program oriented weaving such as in AspectJ [18], the advice quantification [12] here is also user-case oriented. It's quite straightforward that AUL can be used in *product ordering* specification as indicated in Section 2.1. Note the definitions of ADL and AUL are inspired by [11], where non-functional aspects are separated from components themselves to increase the component (and non-functional aspect) reuse, and the non-functional aspects are handled with similar language constructs as ADL and AUL described here.

4.2 Component Description Language (CDL)

CDL is used in the problem space of GDM to specify the components, their required and/or provided services in a way to achieve *maximal combination*, *minimal redundancy*, and *maximum reuse* (as mentioned in Section 3.2) as the result of aspect-oriented generative domain modeling. CDL is defined in TLG as described in Section 3.1.

```

component <componentname>
  <DomainVariable1>,...<DomainVariable-m> ::
    <DomainType-1>; <DomainType-2>;... <DomainType-n>.
  [requires <Domain-Specific-Service>:
    function-call-11, function-call-12,, function-call-1n.]
  [provides <Domain-Specific-Service>:
    function-call-21, function-call- 22,, function-call-n.]
end component <componentname>

```

The first level of CDL provides the type-hierarchy of domain variables. The *requires/provides* specification constitutes the second level. For the *requires* specification, the right-hand side details the requirements; for the *provides* specification, the right-hand specification further specifies the semantics of the provided services.

4.3 Aspectual Component as a Paradigm of Component Assembly

The Aspect Library as shown in Figure 2 captures the *general* business and technology requirements in terms of assembly-related concerns, and a single AUL expression addresses a single concern. In contrast, a component captures groups of behaviors and component assembly captures groups of concerns with mixed scenarios. *Aspectual Component* is used here to address the group of concerns occurring in the component assembly scenario.

The concept of aspectual component¹² is firstly proposed in [22], for which aspects are decoupled from the base program by being defined as a generic aspectual component, which is instantiated later over a concrete data-model using a *connector* construct. Examples of aspectual components and connector specifications will be provided in the following section. The concept of aspectual component fosters the integration between AOSD (Aspect-Oriented Software Development) and Component-Based Software Development (CBSD) ([8], [29]). The aspectual component model will also be used here for component assembly. However, the original *aspectual component* is in Java, while here it is a language-independent specification in TLG. The connector specification classifies server components' related services into a category based on meta-type. The *connector specification* also includes related operations associated with the meta-type. The meta-type can be regarded as one kind of *join point* in AOP, while the related operations in the connector specification provides *advice*. The meta-type in an aspectual component is the basis upon which client and server component get hooked up; the join point model to be used is again type-based as in Section 4.1.

We integrate the ideas into an process diagram in Section 4.3.1, which is reified by an example in Section 4.3.2.

4.3.1 The Overall Picture

Figure 3 provides the multi-stage component assembly process. Stage 1 is mainly about the introduction of the GDM (from domain analysis), which includes CDL in problem space and Aspect Library as configuration knowledge. Stage 2 involves the weaving of the aspect specification into the component specification for each component involved in the assembly process. Stage 3 illustrates the process of the component assembly specification generation based on the aspectual component model. This stage involves a *connector repository*, where the connector specifications will be registered, and the aspectual component will initiate a *query* into the connector repository to find the matching

¹²Note in our own context, the definition of aspectual component is subject to adjustment over its original definition in [22].

connector specification based on meta-type consistency, and to apply the associated advice thereafter. The connector specification is translated from the CDLs of the server component (service provider) and the aspectual component specification is translated from the client component (service consumer). After the full assembly specification is generated, by referencing the component repository (which stores the set of component UMM specifications retrieved by the discovery service in UniFrame), glue/wrapper code will be generated in the final step.

4.3.2 An Example

To help clarify the aforementioned process, a simple example is provided below, demonstrating how the *aspectual component* approach can be adapted to the component assembly process. Assume that the component A is a banking domain client component written in Java RMI requesting some banking service from a server. Below is the partial specification of A's CDL:

```

A.0 Component A
A.1 BankOperation:: Service.
A.2 Bank::BusinessDomain.
A.3 Platform::TechDomain.
A.4 requires BankOperation: Platform= ‘‘RMI’’.
A.5 end Component A.

```

Below is an ADL for a QoS measurement aspect stored in the Aspect Library and AUL to use that aspect.

```

aspect QoSMeter
  advises: BankOperation.
  before: EventTrace.setBeginTime().
  after: EventTrace.setEndTime().
end aspect QoSMeter

apply QoSMeter on A.BankOperation.

```

The above specification of component A weaved with QoSMeter aspects will be translated into the following aspectual component specification:

```

B.0 AspectualCom A
B.1 Bankoperation:: Service.
B.2 Bank::BusinessDomain.

```

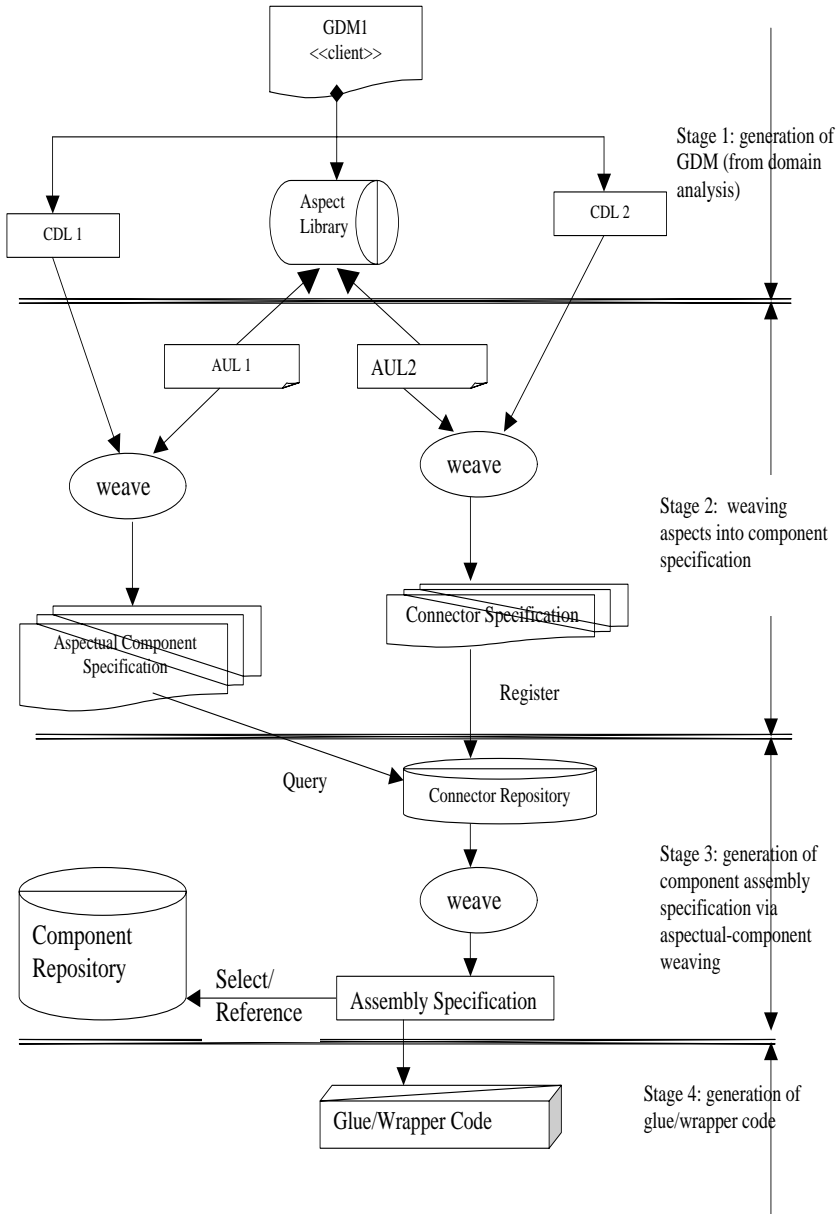


Fig. 3. Multi-Stage Gluing/Wrapping

B.3 expect Bankoperation.

B.4 expect wrap Argument. //usage interface

B.5 replace Bankoperation://modification interface

B.6 EventTrace.setBeginTime(),

```

B.7      expected().wrap(<<Platform='RMI'>>),
          //each <<...>> corresponds to each
          // expression in right hand side of ':' of A4
B.8      EventTrace.setEndTime().
B.9 end AspectualCom A

```

B.6 and B.8 are weaved from the QoSMeter aspect representing client-side concerns. Note those lines prefixed by *expect* denote operation signatures that are expected to be supplied with *advice* (which actually corresponds to server-side services requested), and the *expect*-directive corresponds to the *join points* in AOP. Expected operations are either used (usage interface) or modified (modification interface, preceded with *replace*) in the aspectual component definition. This process is similar to that described in [22].

Assume the component B is a banking domain server component implemented in CORBA providing some banking services.

C.0 Component B.

```

C.1 Withdraw, Deposit:: Port;Bankoperation.
C.2 Bank::Domain.
C.3 Platform::TechDomain .
C.4 provides Bankoperation: Platform= 'CORBA'.
C.5 end Component B.

```

Note in line C.1, the two types denoted in the right hand side of “::” means both withdraw and deposit are not only *Port(s)* (which means they are banking services offered to external components), but also *Bankoperation(s)*.

Below is an ADL for an Access Control aspect [5] from the Aspect Library.

```

aspect AccessControl
  advises: Service.
  before: Log.Check().
end aspect AccessControl

```

This aspect can be applied to any *Service* (meta-type, thus applicable to *Withdraw*). Consequently, before each call to *Service*, *Log.Check()* will be called to verify the credentials.

The following specification will be translated from the component B specification with the AUL of the preceding aspect *AccessControl*.

D.0 connector A-B

```

D.1 {B.Withdraw, B.Deposit} is BankOperation. //join points
D.2 wrap(Argument):
D.3     apply AccessControl on B.WithDraw, B.Deposit,
D.4     apply RMIApect on BankOperation when
D.5     Argument.getname ('Platform')== 'RMI'
D.6 end connector A-B

```

Note that lines D.2-D.5 further implement the *advice* part for the join points (here, *Withdraw* and *Deposit* operations). The body of *wrap* is to wrap the *BankOperation* with RMI specific code. This is similar to [24], in which CORBA related operations are modularized as aspects and then woven into application code to derive a CORBA implementation. The difference here is that, those RMI or CORBA related aspects are pre-built and retrieved from the aspect library, and they are represented with high-level specifications (in ADL) rather than at the application code level. Upon weaving in Stage 4, the *wrap* routine in the connector specification will be woven into the aspectual component specification.

The example illustrated in this section shows that assembly-related concerns (functional and non-functional) of two components can be handled in separate modules (here in the aspectual component definition and connector specification) from the component specification itself. ADL and AUL provide leverage for the assembly process itself to be easily specified and managed. Consequently the assembly can be implemented by using a weaver to weave assembly-specific advice together with component specifications.

5 Discussion

UniFrame, the motivating project of the component assembly approach presented here, aims at automating the process of integrating heterogeneous components to create distributed systems that conform to quality requirements. Generative Programming (GP) is the underpinning solution to fulfill this vision. In order to realize the vision of GP for the highest level of automation, during the domain engineering phase, the creation of the domain model may be applied using Model Integrating Computing (MIC) [20], which is a technology for using domain-specific modeling and a model based generator to compose systems of various forms. MIC has been applied to create a Generic Feature Modeling Environment (GFME) [7] to model system families and generate reusable assets automatically. Based on the component assembly approach presented in this paper, Table 2 describes generative programming in UniFrame.

<i>Generative Programming</i>	<i>UniFrame</i>
Feature modeling	GFME
Components are generated in domain implementation phase	Components are implemented by vendors. Generation only occurs at system level
Configuration Knowledge	Aspect Library
Mapping of problem space to solution space	Resource Discovery Service to search components based on component specification
Domain Specific Language (DSL)	CDL, AUL, ADL
Generator	Aspect Weaver

Table 2
Generative Programming in UniFrame

Also note the assembly paradigm described in Section 4.3 follows a client/server architecture, whereby the client component (service consumer) specification is translated into the aspectual component specification. In the event the components to be assembled are not following that kind of architecture, the *ordering* specification itself may be translated into an aspectual component specification, and then the assembly process as shown in Figure 3 can be applied.

6 Related Work and Conclusion

Recently, there has been work on the application of AOSD to CBSD. One notable work is the aspectual component [22] as described in Section 4.3, which provides a language approach to the effort of reusing aspects. The aspectual component model is adjusted and used here for component assembly. Grundy further introduces the notion of *Aspect-Oriented Component Engineering* (AOCE) ([13], [14], [15], [16]). The aspects in AOCE have a broad definition, which include user interfaces, collaborative work, distribution, persistency, memory management, transaction processing, security, data management, component inter-relationships, and configuration characteristics. AOCE, as an engineering approach, covers the lifecycle of component engineering, from component requirements and specification, to implementa-

tion, deployment, and testing. In contrast to AOP, which highly relies on *code weaving*, AOCE aims to use aspect-codified capacities to support component provisions and requiring of aspect-related services in a general way. In this sense, AOCE can be applied for building the aspect library. None of the related work ever considers applying AOSD to assist the component assembly, however.

This paper presents an approach to apply aspect orientation in the generative domain modeling phase and then leverage the aspect weaver to help component assembly, in particular, for assembling components of client/server architecture. Two repositories (aspect library, connector repository) are used, which aligns with the distributed component assembly style. A type-based join point model is used which can efficiently decouple the aspect definition and aspect usage to promote the reuse of aspects. Compared with the invasive composition approach as described in [1], we weave the assembly-related concerns toward ultimately generating stub/skeleton code for gluing/wrapping components, while the original components (which represent the business logic core), together with their references to stub/skeleton code, will not be affected. This is necessary for black-box components which do not allow invasive methods.

Future work includes the evolution of the aspect library, the application of MIC to domain engineering to automatically generate CDL, and the development of the weaver to weave CDL and ADL. The implementation of glue/wrapper code generation based on the generated assembly specification using the UMM specifications of discovered components must also be integrated into this process.

7 Acknowledgements

We'd like to acknowledge the anonymous reviewers for their helpful suggestions. This research is supported by the U. S. Office of Naval Research under the award number N00014-01-1-0746.

References

- [1] Aßmann, U., "Invasive Software Composition," Springer-Verlag, 2003.
- [2] Brahmamath, G. J., Raje, R. R., Olson, A. M., Auguston, M., Bryant, B. R., Burt, C. C., *A Quality of Service Catalog for Software Components*, Proceedings of the Southeastern Software Engineering Conference ((SE)² 2002), pp. 513-520, April, 2002.
- [3] Brown, A. W., "Large-Scale Component-Based Development," Prentice Hall, 2000.
- [4] Bryant, B. R., Lee, B.-S., *Two-Level Grammar as an Object-Oriented Requirements Specification Language*, Proceedings of 35th Hawaii International Conference on System

- Sciences, 2002,
http://www.hicss.hawaii.edu/HICSS_35/HICSSpapers/PDFdocuments/STDSL01.pdf.
- [5] Burt, C. C., Bryant, B. R., Rajee, R. R., Olson, A. M., Auguston, M., *Model Driven Security: Unification of Authorization Models for Fine-Grain Access Control*, Proceedings of 7th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2003), pp. 159-171, September, 2003.
 - [6] Cao, F., Bryant, B. R., Burt, C. C., Rajee, R. R., Auguston, M., Olson, A. M., *A Translation Approach to Component Specification*, OOPSLA '02 Companion, pp. 54-55, November, 2002.
 - [7] Cao, F., Bryant, B. R., Burt, C. C., Huang, Z., Rajee, R. R., Olson, A. M., Auguston, M., *Automating Feature-Oriented Domain Analysis*, Proceedings of 2003 International Conference of Software Engineering Research and Practice (SERP 2003), pp. 944-949, June, 2003.
 - [8] Choi, J. P., *Aspect-Oriented Programming with Enterprise JavaBeans*, Proceedings of 4th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2000), pp. 252-261, September, 2000.
 - [9] Colyer, A., Blair, G., Rashid, A., *Managing Complexity in Middleware*, Proceedings of the 2nd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), March, 2003.
 - [10] Czarnecki, K., Eisenecker, U. W., "Generative Programming: Methods, Tools, and Applications," Addison Wesley, 2000.
 - [11] Duclos, F., Estublier, J., Morat, P., *Describing and Using Non Functional Aspects in Component Based Applications*, Proceedings of 1st International Conference on Aspect-Oriented Software Development (AOSD 2002), pp. 65-75, 2002.
 - [12] Filman, G., Friedman, D., *Aspect-Oriented Programming is Quantification and Obliviousness*, Proceedings of OOPSLA Workshop on Advanced Separation of Concerns, pp. 168-177, October, 2000.
 - [13] Grundy, J. C., *Multi-perspective Specification, Design and Implementation of Components using Aspects*, International Journal of Software Engineering and Knowledge Engineering, 10(6):713-734, December 2000.
 - [14] Grundy, J. C., *An Implementation Architecture for Aspect-oriented Component Engineering*, Proceedings of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications, pp. 249-256, June, 2000.
 - [15] Grundy, J., Patel, R., *Developing Software Components with the UML, Enterprise Java Beans and Aspects*, Proceedings of the 2001 Australian Software Engineering Conference, pp. 127-136, August 2001.
 - [16] Grundy, J. C., Ding, G., *Automatic Validation of Deployed J2EE Components Using Aspects*, Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE 2002), pp. 47-58, September 2002.
 - [17] Keller, R., Hölzle, U., *Binary Component Adaptation*, Proceedings of European Conference on Object-Oriented Programming (ECOOP'98), Springer-Verlag, LNCS 1445, pp. 307-329, 1998.
 - [18] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., Irwin, J., *Aspect-Oriented Programming*, Proceedings of European Conference on Object-Oriented Programming (ECOOP'97), Springer-Verlag, LNCS 1241, pp. 220-242, 1997.
 - [19] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W., *An Overview of AspectJ*, Proceedings of European Conference on Object-Oriented Programming (ECOOP'01), Springer-Verlag, LNCS 2072, pp.327-353, 2001.
 - [20] Ládeczi, Á., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J. and Karsai, G., *Composing Domain-Specific Design Environments*, IEEE Computer, 34(11):44-51, 2001.

- [21] Lee, B.-S., Bryant, B. R., *Automated Conversion from Requirements Documentation to an Object-Oriented Formal Specification Language*, Proceedings of ACM Symposium on Applied Computing (SAC 2002), pp. 932-936, 2002.
- [22] Lieberherr, K., Lorenz, D., Mezini, M., *Programming with Aspectual Components*, Technical Report, NU-CCS-99-01, 1999,
<http://www.ccs.neu.edu/research/demeter/papers/aspectual-comps/aspectual.ps>.
- [23] Parnas, D., *On the Criteria To Be Used in Decomposing Systems into Modules*, Communications of the ACM, 15(12): 1053-1058, December 1972.
- [24] Pulvermuller, E., Klaeren, H., Speck, A., *Aspects in Distributed Environments*, Proceedings of Generative Component-based Software Engineering (GCSE 99), Springer-Verlag, LNCS 1799, pp. 37-48, September 1999.
- [25] Raje, R., *UMM: Unified Meta-object Model for Open Distributed Systems*, Proceedings of 4th IEEE International Conference of Algorithms and Architecture for Parallel Processing (ICA3PP 2000), pp. 454-465, 2000.
- [26] Raje, R. R., Auguston, M., Bryant, B. R., Olson, A. M., Burt, C. C., *A Quality of Service-Based Framework for Creating Distributed Heterogeneous Software Components*, Concurrency and Computation: Practice and Experience, 14(12):1009-1034, 2002.
- [27] Siram, N. N., Raje, R. R., Auguston, M., Bryant, B. R., Olson, Burt, C. C., A. M., *An Architecture for the UniFrame Resource Discovery Service*, Proceedings of 3rd International Workshop on Software Engineering and Middleware (SEM 2002), Springer-Verlag, LNCS 2596, pp. 22-38, 2002.
- [28] Sutherland, J., Heuvel, W.-J. v. d., *Enterprise Application Integration and Complex Adaptive Systems*, Communications of the ACM, 45(10):59-64, October, 2002.
- [29] Suváe, D., Vanderperren, W., and Jonckers, V., *JAsCo: an Aspect-Oriented approach tailored for Component-based Software Development*, Proceedings. of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003), pp. 21-29, March, 2003.
- [30] Ubayashi, N., Tamai, T., *Aspect-Oriented Programming with Model Checking*, Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002), pp. 148-154, April, 2002.