Using Two-Level Grammar in Component Specification

Fei Cao

[1]Department of Computer and Information Sciences
University of Alabama at Birmingham
1300 University Boulevard
Birmingham, AL 35294, USA
caof@cis.uab.edu

Keyword: UniFrame, Unified Meta-component Model, Two-Level Grammar
Classification: First year's Ph.D. work

1.Introduction

Heterogeneous legacy software systems are ubiquitous. The methodology of integrating existent software systems not only represents a cost-effective solution of software development, but also increases productivity by reducing the time to market. One of the Object Management Group (OMG) initiatives is Model Driven Architecture (MDA) [1]. The vision of MDA includes standards that enable generative construction of interoperating bridges between different technologies leveraging application and platform knowledge.

In a similar vein of OMG's MDA, UniFrame [2] is proposed, which is a framework for seamless interoperation of heterogeneous distributed software components. It is based on the Unified Meta-component Model (UMM) [3] for describing components. Systems constructed by component composition should meet both functional and non-functional requirements such as the Quality of Service (QoS) [4]. UniFrame includes a specification of appropriate QoS parameters, which provide metrics of service at both the component level and system level, so that the software system produced by assembling heterogeneous components can be benchmarked over not only functional requirements, but also non-functional criteria. A Generative Domain Model (GDM) [5] is used to describe the properties of domain specific components and to elicit the rules for component assembly.

In UMM, the Internet Component Broker (ICB) and Headhunters are proposed as two infrastructures in an effort to seamlessly integrate heterogeneous components. ICB provides translation capacity in terms of adapter technology for achieving interoperability, while Headhunters actively detect the presence of new components in the search space, register their functionality and attempt match-making between client components (service requesters) and server components (service providers) based on the GDM of involved components.

We propose a Component Description Language (CDL) using Two-Level Grammar (TLG) [6] as the formalism not only to represent the UMM, QoS, and GDM, but also for specifying the rules of the component assembly process. During the process of component retrieval and assembly, if a component with expected service offerings doesn't satisfy the expected QoS, it is subject to replacement by an alternative component.

2. Related Work

Traditional component specification methods such as UniCon [7] and Wright [8] leverage Architecture Description Languages (ADL) to model software systems. They are typically represented via some formal notations like Z [9] and semantic theory such as communicating sequential processes (CSP) [10] or finite state machines. They have the analysis capacity to predict the properties of the whole system by reasoning over constituent components, which are mostly expressed in an abstract paradigm. But their underlying complex formalism is not convenient for component service exposure, discovery, and the automation of component assembly in a distributed environment. TLG, with a natural-language-like light-weight formalism, is a good candidate to fulfill this goal. XML [11], even though widely used for data representation, is not sufficient to model semantics of a component with a generic Document Type Definition (DTD). It's not the appropriate choice as a specification language to directly model components; the usual practice is to provide a set of API's for writing XML documents, with XML as the underlying vehicle for representing or exchanging data information syntactically.

3. Two-Level Grammar

The name "two-level" of TLG comes from the fact that TLG contains two context-free grammars corresponding to the set of type domains and the set of function definitions operating on those domains.

The type domain declarations have the following form:

Identifier-1,*Identifier*-2, ..., *Identifier*-m::
DataType1; **DataType**2;…; **DataType**-n.

which means that the union of **DataType**-1, **DataType**-2, …, **DataType**-n forms the type definition of *Identifier*-1, *Identifier*-2,… *Identifie*r-m.

The function definitions have the following 3 forms:

function signature.
function signature: **function-call**-1, **function-call**-2,…, **function-call**-n.
function signature: **function-call**-11, **function-call**-12,…, **function-call**-1j;
function-call-21, **function-call**-22,…, **function-call**-2j;
…
function-call-n1, **function-call**-n2,…, **function-call**-nj.

The function body on the right side of ':' specifies the rules of the left hand side function signature. There may be no rules at all, as in the first case. ';' is used in the right hand side to delimit multiple rules which share the same function signature on the left hand side. A detailed explanation of TLG is given in [6].

4. Component Specification

Though based on natural language, TLG is also a formal notation in the sense that it's strictly typed. Its semantics are embodied by rules in the function bodies. Because of the fact that TLG consists of both the type domain definitions and function definitions operating on those domains, TLG may be defined in the form of a class in which case the type domains define the instance variables of the class and the function definitions define the methods of the class. Also components are usually in the Object-Oriented paradigm, so it's quite appropriate that TLG may be used as a meta-language specifying components for component matching at a high level in combination with

domain knowledge, while meta-level specification further facilitates the deployment of components such as generating wrapper/glue code. So TLG will be a cornerstone for constructing a component framework. Moreover, TLG's natural-language style relieves itself from restrictive formal mathematical notations and facilitates the automation leveraging natural language processing technology.

In short, the TLG will be applied in the following two levels:

- Component level

Based on UMM, TLG specifies the following items, namely Computational Attributes, which include Functional Attributes such as Syntactic Contract and Implementation Technology; Cooperative Attributes; Auxiliary Attributes; and QoS. Examples of QoS metrics include Security, Availability, and Throughput (see [4] for details). Component-level specifications are provided by component developers. A simple example is given in Appendix A.

- System level

This level mainly specifies the composition of components. Here we confine our scope to component-component composition, without framework composition. At this level, TLG will be used to specify code assembly rules for the component-level TLG specification.

Figure 1 illustrates the scenario of the above statements:

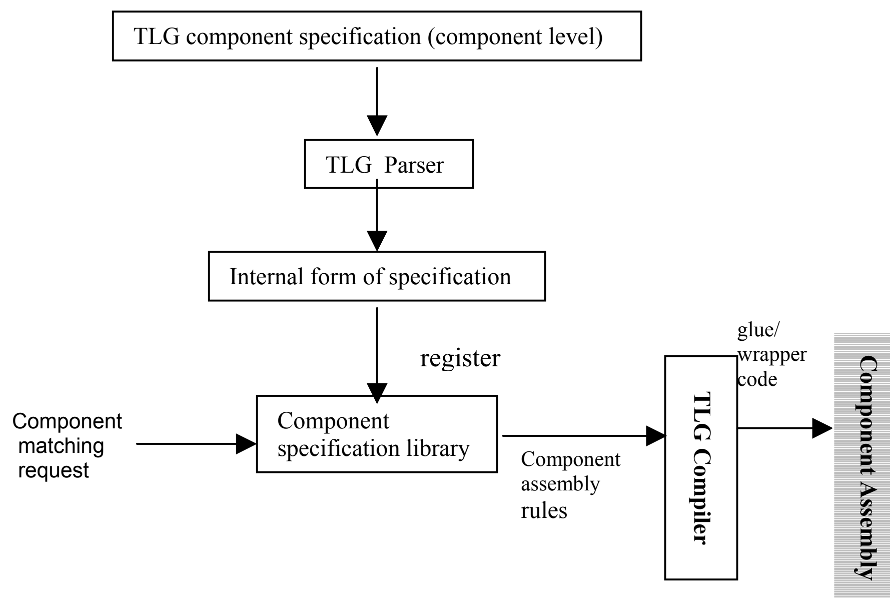

Fig. 1 Application of TLG for Component Assembly

From Fig.1, we can see component-level TLG specifications are validated, then parsed into some internal forms of specification and added to a specification library, which is subject to external retrieval. The TLG compiler will be used to generate the targeted code over the specification library in observance to the assembly rules given in TLG. The component assembly rules are affiliated with the component specification library. An example for illustration is given in Appendix B.

5. Summary and Future Work

In this paper, we explore the application of Two-Level Grammar as a formalism to represent the UMM model and to specify the generative assembly rules for components. We need to have more experience with the interoperability of heterogeneous components to elicit the ultimate CDL specifics. A toolkit for CDL is also under development, which includes automatic CDL generation from a GUI input. A graphical view of CDL in UML is also going to be designed by converting it to XMI format [12]. With the need to further integrate the assembled components, an intermediate form of the assembled components, i.e., TLG specification for the assembled UMM component, may be expected as the output form component assembly. So the TLG compiler may be further developed on demand.

Acknowledgements

This work is supervised by Dr. Barrett R. Bryant (bryant@cis.uab.edu) of the Department of Computer and Information Sciences, University of Alabama at Birmingham and supported in part by the U. S. Office of Naval Research under the award number N00014-01-1-0746.

References

1. Object Management Group (OMG): Model Driven Architecture: A Technical Perspective. Technical Report. Document # ormsc/2001-070-1. Framingham, MA: Object Management Group. July 2001.
2. R. R. Raje, B. R. Bryant, M. Auguston, 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.
3. 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.
4. G. J. Brahnmath, R. R. Raje, A. M. Olson, M. Auguston, B. R. Bryant, and C. C. Burt: A Quality of Service Catalog for Software Components. Proc. $(SE)^2$ 2002, Southeastern Software Engineering Conf..
5. K. Czarnecki, U.W. Eisenecker: Generative Programming: Methods, Tools, and Applications. Addison Wesley, 2000.
6. B. R. Bryant, B.-S. Lee: Two-Level Grammar as an Object-Oriented Requirements Specification Language. Proc. 35th Hawaii Int. Conf. System Sciences, 2002, http://www.hicss.hawaii.edu/HICSS_35/HICSSpapers/PDFdocuments/STDSL01.pdf.
7. M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, G. Zelesnik.: Abstractions for Software Architecture and Tools to Support Them. IEEE Transactions on Software Engineering, v.21 n.4, p.314-335, April, 1995.
8. R. Allen, D. Garlan: Formalizing Architectural Connection. Proc. 16th Int. Conf. Software Engineering, 1994, pp: 71-80.
9. J. M. Spivey: The Z Notation: A Reference Manual. Prentice Hall, 1989.
10. C. A. R. Hoare: Communicating Sequential Process. Prentice Hall, 1985.
11. XML: http://www.w3c.org/XML/.
12. XMI: http://www.omg.org/technology/documents/formal/xmi.htm.

Appendix A: A Component Specification Using TLG

```
component SimpleBank  extends UMMComponent.

           Technology:: String.
           Domain:: String.
           IOR:: String
           Amount :: Float.
           AccountNumber :: String.
           Pin :: String.
           Withdraw:: FunctionName.
           Balance:: FunctionName.
           Deposit:: FunctionName.
     init: Domain ::= "Bank", Technology::="CORBA",
                 IOR::="IOR:000000saji988……98dss3322".
     export Withdraw with Amount AccountNumber
            Pin by classAccount returnType Boolean.

     export Balance with AccountNumber AccountNumber
            Pin by classAccount returnType Float.
     export Deposit with Amount AccountNumber Pin by
            classAccount returnType Boolean.
     lexicon of withdraw: "draw";"subtract".
     lexicon of balance : "check balance".
     lexicon of deposit : "credit";"save".

     /* below is static Quality of Service */
     qos availability : 90%.
            //the duration that a component is available
     qos delay : 10ms.
      //the time between service invocation and completion
end component SimpleBank.
```

Appendix B: A Component System Generation Rule Specification Using TLG

```
     UMM ComponentRepository
           ComponentN::UMMComponent.
           OperationName:: String.
           Package:: StringList.
                 //the header files to be included/imported

           generate java wrapper code ComponentN
               OperationName using CORBA with Package :
                 ComponentN get OperationName !=Empty,
                 ComponentN init,
                 ComponentN get Technology = "CORBA",
     /* "ComponentN get Technology" returns the Technology
     variable of ComponentN, so is the other get operation */
                 ComponentN get IOR!= Empty,
                 // Currently, assume CORBA use IOR for IIOP
           return CorbaClientCode.
     end  UMM ComponentRepository
```