



Formal Specification of Generative Component Assembly Using Two-Level Grammar *

Barrett R. Bryant
Carol C. Burt
Computer/Information Sci.
Univ. Alabama-Birmingham
Birmingham, AL 35294, USA
bryant@cis.uab.edu
cburt@cis.uab.edu

Mikhail Auguston
Computer Science
New Mexico State University
Las Cruces, NM 88003, USA
mikau@cs.nmsu.edu

Rajeev R. Rajeev
Andrew M. Olson
Computer/Information Sci.
Indiana Univ. Purdue Univ.
Indianapolis, IN 46202, USA
rraje@cs.iupui.edu
aolson@cs.iupui.edu

ABSTRACT

Two-Level Grammar (TLG) is proposed as a formal specification language for generative assembly of components. Both generative domain models and generative rules may be expressed in TLG and these specifications may be automatically translated into an implementation which realizes an integration of components according to the principles of the Unified Meta-component Model (UMM) and Unified Approach (UA) to component integration. Furthermore, this implementation realizes Quality of Service (QoS) guarantees by means of static QoS verification at the time of system assembly, and dynamic QoS validation on a set of test cases.

Categories and Subject Descriptors

D.2.1 [Software Engineering]: Requirements/Specifications—*languages, tools*; D.2.11 [Software Engineering]: Software Architectures—*languages*; D.2.12 [Software Engineering]: Interoperability—*distributed objects*

General Terms

Languages

Keywords

Component-based software, formal specification, generative programming, Two-Level Grammar

*This material is based upon work supported by, or in part by, the U. S. Army Research Laboratory and the U. S. Army Research Office under contract/grant numbers DAAD19-00-1-0350 and 40473-MA, and by the U. S. Office of Naval Research under award number N00014-01-1-0746.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SEKE '02, July 15-19, 2002, Ischia, Italy.

Copyright 2002 ACM 1-58113-556-4/02/0700...\$5.00.

1. INTRODUCTION

The recent shift in the focus of OMG (Object Management Group) to “Model Driven Architecture” (MDA) [10] is a recognition that to create mechanized software and bridging of component architectures requires standardization not only of infrastructure but also Business and Component Meta-Models. This emphasizes the fact that a comprehensive meta-model, that seamlessly encompasses heterogeneous components by capturing their necessary aspects including Quality of Service (QoS) and associated guarantees, is needed for creating future generation of distributed systems. The UniFrame project proposes a Unified Meta-component Model (UMM) [11] for distributed component-based systems, and a Unified Approach (UA) [11] for integrating these components. Component development and deployment starts with a UMM requirements specification of a component from a particular domain. This specification is natural language-like and indicates the functional (i.e., computational) and non-functional (i.e., QoS parameters) features of the component. This specification is then refined into a formal specification, based upon the theory of Two-Level Grammar (TLG) [5]. Generative domain models and generative rules for system assembly [6] may be expressed in TLG and these specifications may be automatically translated into an implementation which realizes an integration of components.

2. THE UNIFIED APPROACH

The distinctive features of the Unified Approach are:

- The developer of the desired distributed system presents to this process a system query, in a structured form of natural language, that describes the required characteristics of the distributed system. The query is processed using the domain knowledge (such as key concepts from a domain) and a knowledge-base containing the UMM description of the components for that domain. From this query a set of search parameters is generated which guides “head-hunter” agents for a component search in the distributed environment. Head-hunters serve to locate the components which are needed to complete the requested system [12].
- A set of potential components is collected for that do-

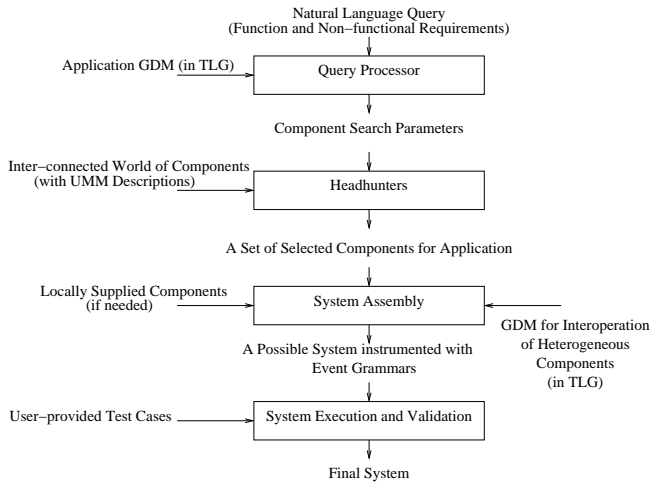


Figure 1: System Assembly in UniFrame

main, each of which meets the Quality of Service (QoS) requirements specified by the developer. QoS requirements are expressed in terms of a catalog of parameters established for this purpose [4]. After the components are fetched, the system is assembled according to the generation rules embedded in the generative domain model. Essentially, the generated code constitutes the glue/wrapper interface between the components.

- Along with the generated system will be a formal UMM specification of the generated system so that it may be used in subsequent assemblies. This formal UMM specification will also be a basis for generating a set of test cases to determine whether or not an assembly satisfies the desired QoS.
- Static QoS parameters (e.g. dependability of the component) are processed during generation time. Dynamic QoS parameters (e.g. response time of the component) result in instrumentation of generated target code based on event grammars [1, 2], which at run time produce the corresponding QoS dynamic metrics which may be measured and validated.

QoS parameters require instrumentation necessary for the run-time QoS metrics evaluation. Based on the query or informal requirements, the user has to come up with a representative set of test cases. Next the implementation is tested using the set of test cases to verify that it meets the desired QoS criteria. If it does not, it is discarded. After that, another implementation is chosen from the component collection. This process is repeated until an optimal (with respect to the QoS) implementation is found, or until the collection is exhausted. In the latter case, the process may request additional components or it may attempt to refine the query by adding more information about the desired solution from the problem domain. If a satisfactory implementation is found, it is ready for deployment. The complete view of this system is shown in Figure 1.

A few attempts have been made to incorporate QoS into component-based software systems. The Aster project [7] uses architectural descriptions of components and their interactions, including non-functional properties, to customize middleware. Quality Objects (QuO) [3], a framework for providing QoS to software applications composed of objects

distributed over wide area networks, bridges the gap between socket-level QoS and distributed object level QoS, emphasizing specification, measuring, controlling, and adapting to changes in QoS. RAPIDware [8], an approach to component-based development of adaptable and dependable middleware, uses rigorous software development methods to support interactive applications executed across heterogeneous networked environments. Process^{NFL} [9] is a language for describing non-functional properties of software, which may include QoS properties. The Unified Approach is concerned not only with specifying QoS properties of components, but also to assure satisfaction of these properties in an implementation resulting from assembling the components. It should be noted that the assurance of QoS (as described above) indicates that a component can guarantee appropriate values for its QoS parameters in an ‘ideal’ situation. This does not guarantee that a component will be able to either provide this QoS under failure circumstances or will automatically adjust its QoS to hide the failures. For the failure situations, the ideas provided by Aster, QuO, or RAPIDware can be incorporated.

3. TWO-LEVEL GRAMMAR SPECIFICATION

Two-Level Grammar (TLG) is a formal notation based upon natural language and the functional, logic, and object-oriented programming paradigms. The “two levels” are two context-free grammars defining the set of type domains and the set of function definitions operating on those domains, respectively. These grammars may be defined in the context of a class in which case type domains define instance variables of the class and function definitions define methods of the class. The TLG formalism is used to specify the generative rules needed for component assembly and the output of the TLG will provide the desired target code (e.g., glue and wrappers for components and necessary infrastructure for distributed run-time architecture). All of this is implemented according to the process for translating TLG specifications into executable code [5].

We illustrate the formal specification of generative rules using TLG by means of a simple bank account management system. The specification of a bank account should include its attributes and the operations it should perform, such as `checkBalance`, `deposit`, or `withdraw`. Assume that the GDM in this example contains a rule for system assembly that specifies that a Bank Account Management System consists of one of each of the two component types, *AccountServer* and *AccountClient*, each of which follows the bank account feature model. Further, let there be two instances of *AccountServer* and one instance of *AccountClient*. Server components are heterogeneous – `JavaAccountServer` adheres to the Java-RMI model and has methods `javaDeposit`, `javaWithdraw`, and `javaBalance`, and QoS parameters *Availability* $\geq 85\%$ and *Response Delay* $< 30\text{ ms}$; while `CorbaAccountServer` uses the CORBA model and has methods `corbaDeposit`, `corbaWithdraw`, and `corbaBalance`, and QoS parameters *Availability* $\geq 90\%$ and *Response Delay* $< 10\text{ ms}$. The client, `JavaAccountClient`, is developed by using the Java-RMI model, with calls to server `depositMoney`, `withdrawMoney`, and `checkBalance`, and QoS parameters *Availability* $\geq 90\%$ and *Response Delay* $< 50\text{ ms}$. The goal is to assemble a bank account management system from

these available components.

Queries are stated in a structured form of natural language. The general form of a query is to request creation of a system that has certain QoS parameters. The name of the system is important in identifying the application domain and the QoS parameters should also follow the catalog standards. A sample query for the above example can be informally stated as: *Create a bank account management system that has availability $\geq 50\%$ and response delay < 100 ms.* This query requires the satisfaction of one static and one dynamic QoS parameter. From the query and the available knowledge in the GDM associated with bank account management systems, a query will be formulated for a headhunter in the UMM. In response, the headhunter will discover the three components and their QoS properties. Note that the availability QoS parameter is used to screen potential components at the time they are retrieved. The catalog specification for this parameter suggests that the availability criteria should be multiplied, so the availability of the Java-Java system is 76.5% and for the Java-CORBA system 81%, both meeting the stated criteria.

TLG is used as the formalism for both the UMM and generative rules. The UMM formalization establishes the context for which the generative rules may be applied. TLG functions include generative rules for construction of wrapper/glue code and event grammar instrumentation to assure the QoS of the bank account record management system. The GDM for bank account management systems will be described according to this template, including both generation rules and QoS parameter processing.

A sampling of TLG rules which may be used to generate the appropriate glue/wrapper code to connect the components of the bank account management system is presented below. These rules are based on selecting from the GDM for bank account management systems the appropriate system model for this two-component DCS. The generation rule to produce Java code for two UMM models representing a client and server, respectively, is expressed using a TLG function which has a signature followed by a set of sub-functions to be executed when the main function is called. Function keywords are indicated in bold while class/object names are italicized.

```
generate system from ClientUMM and ServerUMM :
  ClientOperations := ClientUMM get operations,
  ServerOperations := ServerUMM get operations,
  OperationMapping :=
    map ClientOperations into ServerOperations,
  ComponentModel :=
    ServerUMM get component model,
  generate java code for OperationMapping
  using ComponentModel.
```

The main tasks are to map client operations onto server operations, e.g., `depositMoney` in `JavaAccountClient` maps to `corbaDeposit` in `CorbaAccountServer` or to `javaDeposit` in `JavaAccountServer`, and then generate the code to implement this mapping. The next set of rules describes the specifics of generating CORBA code in Java to implement the mapping that arises by integrating the `JavaAccountClient` with the `CorbaAccountServer`, including the mechanism for generating individual methods. The generated code is distinguished from types (variables) and function keywords by using a typewriter font.

```
generate java code OperationMapping using corba :
  CorbaPackageName :=
    OperationMapping get corba package name,
  CorbaObjectClass :=
    OperationMapping get corba object type,
  ClassName := OperationMapping get class name,
  JavaClassName := Java || ClassName,
  CorbaObjectName := object || ClassName,
  SetUpCode := ComponentModel generate java code,
  Operations :=
    generate java code for OperationMapping,
  return
  import CorbaPackageName . *;
  public class JavaClassName {
    private CorbaObjectClass CorbaObjectName ;
    // initialize CORBA client module
    public void init () {
      SetUpCode
    }
    Operations
  }.
}
```

The class structure required by the Java implementation consists of a function `init` to set up the CORBA ORB and the operations needed in the server. This includes the code to initialize the CORBA object so that future operations can refer to it. It is necessary to first extract the names of the CORBA package, class of the CORBA object to be referenced within the package, and the name of the class itself. These are all stored in the *OperationMapping*. The name of the Java class generated is simply the string "Java" concatenated¹ with the name of the server class, i.e., `JavaCorbaAccountServer`. The name of the CORBA object is generated in a similar way. For simplicity, only the case where the class is to contain a single method is shown. Multiple methods are handled similarly.

```
generate java code for
  OperationName1 ArgumentList1 ReturnType
  maps to
  OperationName2 ArgumentList2 ReturnType :
  JavaReturnType := java type of ReturnType,
  JavaArgumentList :=
    list all ArgumentList1
    mapped to JavaArgument
    by function java argument of
      Argument is JavaArgument,
  JavaArgumentListDefinition :=
    separate JavaArgumentList by , ,
  OperationCall := generate java code for
  OperationName2 ArgumentList1 ReturnType,
  return
  public JavaReturnType OperationName1
    ( JavaArgumentListDefinition ) {
    EventTrace . setBeginTime ();
    OperationCall
    EventTrace . setEndTime ();
    EventTrace . calculateResponseTime ();
  }.
}
```

¹The TLG concatenation operation (||) differs from juxtaposition in that it does not produce a space between the operands.

This generation assumes that the methods have the same return type and so the main task is to express the arguments of the first operation in terms of Java syntax, generate the appropriate method call, and instrument the code with the event grammar mechanism to measure the response time. The former is accomplished by using a TLG list comprehension to map the arguments in *ArgumentList1* into corresponding Java arguments represented by *JavaArgumentList*. Each *Argument* from *ArgumentList1* is mapped into a *JavaArgument* using the function **java argument of Argument is JavaArgument**. There is a subtlety here in that *JavaArgumentList* is an abstract syntax representation of the desired argument list and so this must be made into concrete syntax using the **separate** operation which adds the appropriate commas in between the argument declarations. The appropriate method call is handled by the rule below.

generate java code for

```

OperationName ArgumentList Return Type :
IdList := list all Argument from ArgumentList
mapped to Id by
function argument id of Argument is Id,
IdListInCall := separate IdList by , ,
return CorbaObjectName . OperationName
( IdListInCall );

```

Again a list comprehension is used to extract the arguments from the argument list, this time only the identifier part (achieved by **function argument id of Argument is Id**). Likewise, the abstract syntax representation must be made concrete by comma separators.

Finally, the event grammar instrumentation is added to measure the time at the beginning of the server method call and again at the end so that the actual response time can be evaluated against the required QoS ($< 100ms$). The QoS metrics for "response delay" mean execution time for each method call within the server or client, and require the instrumentation of each generated wrapper for the client/server method call with auxiliary functions able to check the clock at the beginning and at the end of method call, calculate the duration, and submit it to the execution monitor (also generated as a part of instrumentation). We assume that these are taken care of by a class called *EventTrace*. Each of the two example systems will be implemented with the code for carrying out event trace computations according to test cases which must be supplied by the user. These test cases will be executed to verify that the bank account management system satisfies the QoS specified in the query. If the system is not verified, it is discarded. This verification process is carried out for each of the generated bank account management system (two in the above example). Then the one with the best QoS is chosen, in the above example the *CorbaAccountServer* and *JavaAccountClient* combination.

For the example, the following code for the *depositMoney* function would be produced.

```

public void depositMoney (float ip) {
    EventTrace . setBeginTime ();
    objectCorbaAccountServer . deposit (ip);
    EventTrace . setEndTime ();
    EventTrace . calculateResponseTime ();
}

```

In the future, the efficient generation and update of a distributed computing system will require at least a semi-

automatic integration of software components, based on their advertised QoS, in such a way that it meets the QoS constraints specified by the user. UniFrame facilitates semi-automatic construction of such a system. A simple case study is provided in this paper for illustration, but the principles of the proposed approach can be applied to larger applications.

4. REFERENCES

- [1] M. Auguston. Program behavior model based on Event Grammar and its application for debugging automation. In *Proc. 2nd Int. Workshop Automated and Algorithmic Debugging*, pages 277–291, 1995.
- [2] M. Auguston. Tools for program dynamic analysis, testing, and debugging based on event grammars. In *Proc. SEKE 2000, 12th Int. Conf. Software Engineering Knowledge Engineering*, pages 159–166, 2000.
- [3] BBN Corporation. *Quality Objects (Quo)*, <http://www.dist-systems.bbn.com/tech/Quo>, 2001.
- [4] G. J. Brahmamath, R. R. Raje, A. M. Olson, M. Auguston, B. R. Bryant, and C. C. Burt. A quality of service catalog for software components. In *Proc. (SE)² 2002, Southeastern Software Engineering Conf. (to appear)*, 2002.
- [5] B. R. Bryant and B.-S. Lee. Two-Level Grammar as an object-oriented requirements specification language. In *Proc. 35th Hawaii Int. Conf. System Sciences*, 2002.
- [6] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
- [7] INRIA-Rocquencourt. *ASTER: Software Architectures for Distributed Systems*, <http://www-rocq.inria.fr/solidor/work/aster.html>, 2001.
- [8] Michigan State University. *RAPIDware: Component-Based Development of Adaptable and Dependable Middleware*, <http://www.cse.msu.edu/rapidware>, 2001.
- [9] N. S. Rosa and P. R. F. Cunha and G. R. R. Justo. process^{nf}: A language for describing non-functional properties. In *Proc. 35th Hawaii Int. Conf. System Sciences*, 2002.
- [10] Object Management Group (OMG). Model Driven Architecture: A technical perspective. Technical report, OMG Document No. ab/2001-02-01/04, February 2001.
- [11] R. R. Raje, M. Auguston, B. R. Bryant, A. M. Olson, and C. C. Burt. A unified approach for the integration of distributed heterogeneous software components. In *Proc. Monterey Workshop Engineering Automation for Software Intensive Systems*, pages 109–119, 2001.
- [12] N. N. Siram, R. R. Raje, B. R. Bryant, A. M. Olson, M. Auguston, and C. C. Burt. An architecture for the UniFrame Resource Discovery Service. In *Proc. SEM 2002, 3rd Int. Workshop Software Engineering Middleware (to appear)*, 2002.