A Quality of Service-based Framework for Creating Distributed Heterogeneous Software Components

Rajeev R. Raje[1] Mikhail Auguston[2] Barrett R. Bryant [3] Andrew M. Olson[1] Carol Burt[4]

Abstract

Component-based software development offers a promising solution for taming the complexity found in today's distributed applications. Today's and future distributed software systems will certainly require combining heterogeneous software components that are geographically dispersed. For the successful deployment of such a software system, it is necessary that its realization, based on assembling heterogeneous components, not only meets the functional requirements, but also satisfies the non-functional criteria such as the desired QoS (quality of service). In this paper, a framework based on the notions of a meta-component model, a generative domain model and QoS parameters is described. A formal specification based on Two-Level Grammar is used to represent these notions in a tightly integrated way so that QoS becomes a part of the generative domain model. A simple case study is described in the context of this framework.

Keywords: Distributed systems, Quality of Service, Generative Domain Models, Heterogeneous Components, Formal methods, Two-Level Grammar.

1 Introduction

In the recent past, component-based software design has emerged as a viable and economical alternative to the traditional software design process. The notion of independently created and deployed components, with public interfaces and private implementations, loosely integrating with one another to realize a software solution is appealing. It is even more so in the field of distributed computing, where the underlying heterogeneity can be masked by the use of a coalition of distributed software components. Due to the inherent complexities of the distributed computing paradigm and due to the nascent nature of the component-based approach, the potential of this approach has yet to be fully exploited. Many challenging issues need to be addressed in order to fully harness the potential of the component-based approach to distributed systems. The prominent ones are: a) the creation of a formal meta-component model, b) a mechanism to precisely describe the meta-model and associated features (including the generative rules), c) the formalization of QoS (Quality of Service) offered by components, and d) a mechanism to assure the specified QoS. Thus, a comprehensive framework that will encompass these issues and aid the software developers is needed. In the paper, one such framework is proposed along with its application to a case study.

The rest of the paper is organized as follows. The next section contains a brief description of the related efforts. It is followed by the details of Unified Meta-component Model and Generative Domain Model. The QoS part of the framework is described in section 4. The section that follows discusses formal specification methods used in the proposed framework. A simple case study is described in section 6 and is followed by the conclusion.

2 Related Work

2.1 Component Models

Although component-based software development and its application to the distributed computing are relatively new concepts, a plethora of models and projects have been proposed by academia and industry, e.g., Java™ Remote Method Invocation (RMI) [24], Common Object Request Broker Architecture (CORBA™) [24, 29],

[1]Department of Computer and Information Science, Indiana University Purdue University Indianapolis, 723 W. Michigan Street, SL 280, Indianapolis, IN 46202, USA, {rraje, aolson}@cs.iupui.edu, +1 317 274 5174/9733

[2]Computer Science Department, Naval Postgraduate School, 833 Dyer Rd., SP 517, Monterey, CA 93943, USA, auguston@cs.nps.navy.mil, +1 831 656 2509 – On leave from Computer Science Department, New Mexico State University, USA.

[3]Department of Computer and Information Sciences, The University of Alabama at Birmingham, 1300 University Blvd., Birmingham, Alabama 35294-1170, USA, bryant@cis.uab.edu, +1 205 934 2213

[4]2AB, Inc., 1700 Highway 31, Calera, AL 35040, USA, cburt@2ab.com, +1 205 621 7455

Distributed Component Object Model (DCOM^{TM}) [21, 24], Web-component model/DOM [18], Pragmatic component web [9], Hadas [14], Infospheres [7], Legion [31], and Globus [11]. Some of these are language-centric (RMI), while others allow a limited interoperability (CORBA). Some are general-purpose (DCOM), i.e., not concentrating on any particular application domain, while others are domain-dependent (Legion). However, almost all of these models do not assume the presence of others. Thus, the interoperability which they provide is limited mainly to the underlying hardware, operating system and/or implementation languages. If component-based distributed software systems are to become successful, then there is certainly a need for an approach that will transcend this limited interoperability. One possible approach to achieve comprehensive interoperability is that of using a meta-model for heterogeneous distributed components.

2.2 Generative Programming

In [8] the Generative programming paradigm is defined as: "Generative Programming is about manufacturing software products out of components in an automated way. It requires two steps: a) a design and implementation of a generative domain model, representing a family of software systems (development for reuse), this model includes also domain-specific software generator; b) given a particular requirements specification, a highly customized and optimized end-product can be automatically manufactured from implementation components by means of generation rules (development with reuse)". The notion of generative programming is incorporated in the proposed approach as described in the section 3.2.2.

2.3 Quality of Service (QoS)

Although QoS and its guarantees have been widely used in networking, not many attempts have been made to incorporate QoS into component-based software systems. Quality Objects (QuO) [5] is a framework for providing QoS to software applications composed of objects (especially CORBA-based objects) that are distributed over wide area networks. QuO bridges the gap between the socket-level QoS and the distributed object level QoS. QuO's emphasis is on specification, measuring, controlling, and adapting to changes in QoS. RAPIDware [20] is an approach for component-based development of adaptable and dependable middleware. It uses rigorous software development methods to support interactive applications executed across heterogeneous networked environments. It focuses on specification, design, and use of component-based middleware.

3 Unified Meta-Component Model and Generative Domain Model

3.1 Why a Meta-model?

Given the plethora of component-based models and noting the fact that components, by definition, are independent of the implementation language, tools and the execution environment, it is necessary to answer the questions: *why is a meta-model needed for a seamless interoperation of distributed heterogeneous components?* and *how would a meta-model assist in seamlessly integrating distributed heterogeneous software components?* The answer to these question lies in: a) in any organization, software systems undergo changes and evolutions, b) local autonomy is an inherent characteristic of today's geographically (or logically) dispersed organizations, and c) if reliable software needs to be created for a distributed computing system (DCS) by combining components, then the QoS offered by each component needs to become a central theme of the software development approach.

The consequence of constant evolutions and changes is that there is a need to create prototypes rapidly and experiment with them in an iterative manner. Thus, there is no alternative but to adhere to cyclic (manual or semi-automatic) component-based software development for a DCS. However, the solution of decreeing a common COTS environment, in an organization, is against the principle of local autonomy. Hence, the development of a DCS in an organization will, most certainly, require creating an ensemble of heterogeneous components, each adhering to some model. Also, every DCS is designed and developed with a certain goal in mind, and usually that goal is associated with a certain perception of the quality (as expected from the system) and related constraints.

Thus, there is a need for a comprehensive meta-model that will seamlessly encompass existing (and future) heterogeneous components by capturing their necessary aspects, including the quality of service and associated guarantees offered components. As distributed systems are becoming omni-present and many of them are mission-critical, their software development should emphasize and integrate the QoS-oriented theme.

For enterprise component solutions, the standards necessary to design systems using a meta-model that can be realized in many diverse technologies is an area where significant standards work is now focused. The recent shift in focus for the OMG to "Model Driven Architecture" (MDA) [23] is a recognition that to create mechanized software for the collaboration and bridging of component architectures will require standardization not only of infrastructure but also Business and Component Meta-Models. The need to support the evolution of component models and to describe the capabilities of the models will be key to realizing the full potential of an E-business economy.

3.2 Unified Meta-component Model (UMM) and Unified Approach (UA)

In [26, 27] a unified meta-component model (UMM) and a unified approach (UA) based on it, for distributed component-based systems, is proposed. A brief description of UMM and UA is presented below. A more detailed discussion of UMM and UA is found in [26, 27].

3.2.1 UMM

The core parts of the UMM are: *components*, *service and service guarantees*, and *infrastructure*. The innovative aspects of the UMM are in the structure of these parts and their inter-relations. UMM provides an opportunity to bridge gaps that currently exist in the standards arena. For example, the CORBA Component Model (CCM™) [22] and Java Enterprise Edition component models (J2EE™) are consistent, and yet, because of the absence of a formal meta-model, it is difficult during the evolution of each to recognize when the boundaries that maintain the consistency are crossed. Similarly, it has been demonstrated in numerous products that the Component Object Model (COM™) [28] and CORBA component models are similar (in an abstract sense) enough to allow meaningful bridging. It is, however, not possible to point to a Meta-model that constrains the implementations of these technologies so that bridging is assured in practice.

Component

In UMM, components are autonomous entities, whose implementations are non-uniform, i.e., each component adheres to a distributed-component model but there is no notion of an unified implementational framework. Each component has a state, an identity, a behavior, a well-defined interface and a private implementation. In addition, each component has three aspects: 1) computational, 2) cooperative, and 3) auxiliary.

The computational aspect reflects the task(s) a component carries out. In a DCS, components must be able to 'understand' the functionality of other components. Thus, each UMM component supports the introspection, by which it precisely describes its services to others. UMM takes a mixed approach to indicate the computational aspect of a component – a simple textual part, called *inherent attributes* and a formal precise part, called *functional attributes*. The inherent attributes contain the book-keeping information about a component (e.g., author, version, etc.); while the functional part is formal and indicates precisely the computation, its associated contracts and the level(s) of service the component offers. Both the inherent and functional attributes are specified by the component's creator.

In UMM, components are always in the process of cooperating with each other. This is depicted in the cooperative aspect of each component. Informally, the cooperative aspect of a component contains: i) Pre-processing collaborators – other components on which this component depends, and ii) Post-processing collaborators – other components that may depend on this component.

In addition to computation and cooperation, mobility, security, and fault tolerance are necessary features of a DCS. The auxiliary aspect of a component addresses these features. In UMM, each component can be potentially mobile. The mobility of the component is indicated as a mobility attribute. Similarly, the security and fault-tolerant attributes of a component contain the necessary information about its security and fault-tolerance features.

Service and Service Guarantees

A service offered by a component could be an intensive computational effort or an access to underlying resources. In a DCS, it is natural to expect several choices for obtaining a specific service. Thus, each component must be able to specify the quality of the service offered.

The QoS offered by each component depends upon the computation it performs, the algorithm used, its expected computational effort, required resources, the motivation of the developer, and the dynamics of supply and demand. The QoS is an indication given by an component, on behalf of its owner, about its confidence to carry out the required services. The task of guaranteeing the necessary QoS is a key issue in any quality-oriented framework. Section 4 discusses the solutions provided by the unified approach based on UMM.

Infrastructure

Because local autonomy is inherent in a DCS, forcing every component developer to abide by certain rigid rules is doomed to fail. UMM tackles the issue of non-uniformity with the assistance of the *head-hunter* and *Internet Component Broker*. These are responsible for allowing a seamless integration of different component models and sustaining cooperation among heterogeneous (adhering to different models) components.

The tasks of head-hunters are to detect the presence of new components in the search space, register their functionalities, and attempt match-making between service producers and consumers. A head-hunter is analogous to a binder or a trader in other models, with one difference – a trader is passive, while a head-hunter is active. It attempts at discovering components and registering them. During the registration process, a component informs the head-hunter about its aspects to be used during the matching process. A component may register with multiple head-hunters. Head-hunters may cooperate with each other in order to serve a large number of components.

Considering the heterogeneous nature of the components, it is conceivable that the software realization of a distributed system will require an ensemble of components adhering to different models. This requires a mediator, the *Internet Component Broker*, that will facilitate cooperation between heterogeneous components.

The Internet Component Broker (ICB) acts as a translator between two heterogeneous components. ICB utilizes adapter technology, each adapter component providing translation capabilities for specific models. Thus, an adapter component's computational aspect indicates the models for which it provides interoperability. It is expected that brokers are pervasive in an Internet environment, thus providing a seamless integration of disparate components. Adapter components register with ICB and indicate their specializations (which component models they can bridge efficiently). During a request from a seeker, the head-hunter component not only searches for a provider, but also supplies the necessary details of an ICB.

The adapter components achieve interoperability using the principles of *wrap* and *glue* technology [17]. Wrappers provide a common message-passing interface for components that frees developers from the error prone tasks of implementing interfaces and data conversions. The glue schedules time-constrained actions and carries out the actual communication between components. The automatic generation of glue and wrappers based on component specifications provides a reliable, flexible and cost-effective ways to achieve interoperability.

The functionality of the ICB is analogous to that of an object request broker (ORB). The ICB provides the capability to generate the glue and wrappers necessary for components implemented in diverse component models (and providing service guarantees) to collaborate across the Internet; the ORB does this only at the level of objects written in different programming languages. An ORB defines language mappings and object adapters. An ICB provides component mappings and model adapters. While the ICB conceptually provides the capabilities of existing bridges (COM-CORBA for example), it has key features that are unique; it is designed to encompass all the aspects of components and the QoS features and associated guarantees. Thus, the ICB, in conjunction with head-hunters, provides an infrastructure necessary for scalable, reliable, and secure collaborative computation for a DCS.

3.2.2 UA

The UA is based on the principles of UMM. The creation of a software solution for a DCS, using UA, has two levels: a) component level – developers create components, test and validate the appropriate QoS and deploy the components on the network, and b) system level – a collection of components, each with a specific functionality and QoS, and a semi-automatic generation of a software solution for the particular DCS is achieved. These two levels and associated processes are described below.

Component Development and Deployment Process

The component development and deployment process starts with a UMM requirement specification of a component from a particular domain. This specification is in a natural language and indicates the functional (i.e., computational, cooperative and auxiliary aspects) and non-functional (i.e., QoS constraints) features of the component. This specification is then refined into a formal specification. The refinement is based upon the theory of Two-Level Grammar (TLG) [32] and natural language specifications [6]. The refinement is achieved by the use of conventional natural language processing techniques (e.g. [15]) with a domain knowledge base. TLG specifications allow for the generation of the interface (possibly multi-level) for a component. This interface incorporates all UMM-aspects of a component. The developer then provides the implementation to all the methods indicated in the interface. This process is followed by the validation against requirement specifications. If the results are satisfactory then it is deployed on the network and is discovered by one or more head-hunters.

If the component does not meet the requirement specifications then the developer refines either the UMM requirement specification or the implementation and the cycle repeats.

Formal Specification of Components in UMM

Since the UMM specifications are informally indicated in a natural language like style, UA aims at translating these into more formal specifications using TLG. TLG is a formal notation based upon natural language and the functional, logic, and object-oriented programming paradigms. The reason that TLG is chosen is that it allows queries over the knowledge base to be expressed in a natural language like manner which is consistent with the way in which UMM is expressed. TLG is then a framework under which natural language may be used to both describe and inquire about the nature of components and systems. More details of TLG, which facilitate a formal specification of components and queries, are described in the section 5.

Automated System Generation

In general, different developers will provide on the Internet a variety of possibly heterogeneous components oriented towards a specific problem domain. Once all the components necessary for implementing a specified distributed system are available and a specific problem is formulated, then the task is to assemble them into a solution. The proposed framework takes a pragmatic approach, based on Generative Programming [4, 8], to component-based programming. It is assumed that the generation environment will be built around a generative domain-specific model (GDM) supporting component-based system assembly. The distinctive features of the proposed approach are as follows:

1. 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 (in the form of a TLG) 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.
2. The framework, with the help of the infrastructure, collects a set of potential components for that domain, each of which meets the QoS requirement specified by the developer. 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. The TLG formalism is used to describe the generative rules (see section 6 for further discussion) 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).
3. 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 a assembly satisfies the desired QoS.
4. The static QoS parameters are processed during generation time and hence will be processed by the TLG directly. Dynamic QoS parameters result in instrumentation of generated target code based on event grammars, which at run time will produce the corresponding QoS dynamic metrics.

To summarize, the inputs for the system assembly and generation step are: the query for the system build, UMM descriptions of the components found by headhunters, and the QoS parameters for the system build. The outputs are the generated code instrumented for the dynamic QoS metric evaluation and auxiliary code needed to compile, assemble and run the system, and UMM description of the generated system which makes it possible to add the new component to the component database. Two-level Grammar is the formalism for representing UMM's, GDM's, QoS parameters, supporting queries, and generation rules. Only the queries that have counterparts in the GDM are processed. The GDM contains generation rules for system assembly from the components. The query language becomes an essential part of the proposed approach since the query provides the input for component search via the headhunter mechanism and following glue and wrapper generation. The query supplies the initial parameters for the headhunters to search in the distributed environment and gives the input for the generation step itself.

The proposed approach to the Generative Programming besides the domain-specific generative models involves yet another dimension: components and their attributes found in the distributed environment. Since the environment is changing, the results of a query depends on the component resources available. The attributes found in the UMM descriptors of the fetched components determine the hierarchy of generation rule calls and hence the architecture of the assembled system. This implies that UMM descriptor has to be generation-oriented,

i.e. contains attributes specific for the generation needs. The generation rules represent typical design patterns for the selected domain and more general software design patterns, e.g. as advocated in [10].

QoS parameters given in the query provide yet another aspect for the generated code - the instrumentation necessary for the run-time QoS metrics evaluation. Static QoS parameters are processed at generation time by corresponding rules within the domain model. Since dynamic QoS metrics can be calculated only for particular inputs, in order to find the best possible approximation for the system, the following approach is suggested. 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 same GDM is used to generate the final optimized version of the required system and UMM description of the system if the system is to be used as a stand-alone component.

4 QoS-based Approach

The UA to assuring the QoS of a DCS is made up of three steps: a) the creation of a catalog for QoS parameters (or metrics), b) a formal specification of these parameters, and c) a mechanism for ensuring these parameters, both at each individual component level and at the entire system level. In next few sections, these three steps are described in detail.

4.1 A Catalog of QoS Parameters

There are many possible QoS parameters that a component (and its developer) can use to indicate the associated service. In UA, as a first step, a catalog of QoS parameters is created. The format of this catalog is based on that of the design patterns [10] catalog. This catalog provides a vocabulary for a QoS-based approach. A QoS parameter is entered into this catalog only if it is completely different from the existing ones and appears in many application domains. It is expected that this catalog will gradually evolve over a span of time.

The goal of creating the QoS catalog is two-fold: a) it assists the component developer (or the system integrator) in selecting the necessary QoS parameters for the component (or system) under construction, and b) it enables the developer (or integrator) to ensure the necessary QoS guarantees by integrating the selected QoS parameters into the assurance process.

4.1.1 Description of QoS Parameters

Each parameter is described by using the following features:

1. Name: indicates the name of the parameter.
2. Intent: indicates the purpose of the parameter.
3. Description: provides a brief informal description of the parameter.
4. Influencing Factors: depicts the factors on which the parameter depends along with their measures and degree of influence, if any.
5. Measure: indicates the unit in which to measure the parameter.
6. Known Usages: describes the known usages of the parameter.
7. Aliases: indicates other prevalent names, if any.
8. Related Parameters: indicates other related QoS parameters.
9. Consequences: indicates the effects if this parameter is used in describing the QoS of a component.
10. Levels: indicates possible QoS levels offered by a component.
11. Technologies: indicates the underlying technologies.
12. Applications: indicates the application domains in which the parameter has been used.
13. Exceptions: indicates the possible error situations and associated exception handling capabilities.
14. Example Scenario: indicates a possible scenario where it is appropriate for the parameter to be used.

4.1.2 List and Brief Description of QoS Parameters

In [34], a few QoS parameters for objects are described. That list has been augmented to create a current version of the catalog that contains the following parameters:

1. Throughput: indicates the efficiency or speed of a component (e.g., user-interaction component).
2. Capacity: indicates the maximum number of concurrent requests a component can serve (e.g., server component).
3. End-to-End Delay: indicates the time difference between the invocation of a method of a component to its completion (e.g., numerical computational component).
4. Parallelism Constraints: indicates whether a component can support synchronous or asynchronous invocations (e.g., server component).
5. Availability: indicates the duration when a component is available to offer a particular service (e.g., classifier component).
6. Ordering Constraints: indicates the order of the return results and its significance (e.g., transaction component).
7. Error Rate: indicates the probability of returning incorrect result or no result at all (e.g., arithmetic computational component).
8. Security: indicates the security-related details of a component (e.g., e-commerce component).
9. Transmission: indicates the quality of the data communication provided by a component (e.g., a routing component).
10. Adaptivity: indicates how a component can adapt to changing environment (e.g., information service provider component).
11. Evolvability: indicates how easily a component can evolve over a span of time (e.g., text-editor component).
12. Reliability: indicates reliability of the service offered by a component (e.g., real-time controller component).
13. Stability: indicates whether a component can provide a predictable quality (e.g., network controller component).
14. Result: indicates quality of the results returned (e.g., numerical computational component).
15. Achievability: indicates if a component can provide a higher degree of service than promised (e.g., multimedia transmission component).
16. Priority: indicates if a component if a component is capable of providing prioritized service (e.g., scheduling component).
17. Compatibility: indicates if a component is environment (e.g., platform) dependent or not (e.g., applet component).
18. Presentation: indicates the presentation aspects of the result returned by a component (e.g., database component).

4.1.3 Detailed Sample Description

Although, all the above mentioned parameters have been fully described in [25], for the sake of brevity below only one parameter, *Throughput*, is described in detail.

- Name: Throughput.
- Intent: This parameter indicates the speed of efficiency of a component.
- Description: This parameter is used to specify the number of methods or requests that a component can serve per a given time unit (e.g., second) and the classification of the requested methods based on their read/write behaviors.
- Influencing Factors: This parameter depends on the following factors:
  - Algorithms used by each method and associated complexity measures (e.g., time, space) – weight of this factor is very important.

  - Available resources and their abilities and quantities – weight of this factor is very important.
  - Operating system scheduling scheme – weight of this factor is important.
- Measure: Methods_completed/Second.
- Known Usages: FTP Server, HTTP Server, Email Server, Information Classifying System, User Interaction Environment.
- Aliases: Execution Rate.
- Related Parameters: Capacity, Parallelism Constraints, and End-to-End Delay.
- Consequences: A guarantee of a higher throughput could have an adverse effect on the resources allocated to other components running on that machine thereby deteriorating their performance.
- Levels: The possible levels for throughput could be: a) low (< 50 requests completed per second), b) moderate (< 500 requests completed per second), and c) high (< 5000 requests completed per second).
- Technologies: RPC, RMI, etc.
- Applications: Web, E-commerce, Database, Scientific Computation.
- Exceptions: a) actual throughput is less than the one promised (`LessThanPromisedException`) – this can lead to disastrous situations in critical application domains, and b) actual throughput exceeds the promised number (`MoreThanPromisedException`) – in most cases, this will not have any adverse effect, but in some it can lead to problematic situations.
- Example Scenario: In an information filtering system, a representer component provides the service of converting a textual document into its numerical equivalent form. The representer, typically, supports a function called `represent_document()`. If such a component specifies its QoS as 15 methods_completed/second, then it indicates that the representer is able to convert 15 textual documents into their numerical forms in one second. A representer can also specify that it provides either one level (say 15 methods/second) or two levels (15 methods/sec and 30 methods/sec) or three levels (15 methods/sec, 30 methods/sec and 45 methods/sec) of services.

4.2 QoS Metrics and Implementation

Dynamic QoS metrics can be expressed in a uniform manner based on the system behavior models. In [2, 3] the use of event grammars as a basis for such models is suggested. An event is an abstraction of any detectable action performed during run-time, for instance, execute a statement or call a procedure. An event has beginning, end, and duration, and some other attributes, such as program states at the beginning and end of the event, source code associated with the event, and so on. Two binary relations are defined for the events. One event may precede another event, e.g. one statement execution may precede another, or one event may be included in another, e.g., a statement execution event may appear inside a procedure call event. System execution may be represented as a set of events with the two basic relations between them - event trace. An event grammar is a set of axioms that determines possible configurations of events of different types within the event trace. For example, the axiom

```
execute-assignment : evaluate-expression  perform-destination
```

specifies that the event of the type execute-assignment contains a sequence (with respect to the precedence relation) of events of types evaluate-expression and perform-destination, correspondingly.

Different dynamic QoS metrics could be expressed as appropriate computations over event traces. For example, if 'function-call IS A' denotes an event of the type function call with the name A, then the total duration of this function call may be expressed as:

```
SUM/[ X: function-call IS A FROM execute-program Duration(X)]
```

`[...]` denotes a sequence constructor which selects from the whole event trace (an event of the type execute-program) all events that match the pattern function-call IS A, takes the Duration attribute of those events, and sums them up. Event grammars and the notion of the computations over event traces provide a uniform framework to define different dynamic QoS metrics. This mechanism may be a basis of automatic instrumentation of the generated code.

As has been indicated above, static QoS parameters are processed by generation rules at generation time according to the inference rules encoded in the domain model (see example in section 6.5). Therefore, the event grammar and a language for event trace computations are part of the GDM.

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 QuO [5] or RAPIDware [20] can be incorporated into the UMM and UA.

5 Formal Specification in the Unified Approach

Formal specification in UA is by means of Two-Level Grammar (TLG, also called W-grammar). TLG was originally developed as a specification language for programming language syntax and semantics and was used to completely specify ALGOL 68 [33]. TLG may be used as an object-oriented requirements specification language and also serve as the basis for conversion from requirements expressed in natural language into a formal specification [6]. This section describes the TLG language details that facilitate these processes and elaborate on how the language may be used in formal specification of UMM specifications.

The name "two-level" in Two-Level Grammar comes from the fact that TLG consists of two context-free grammars interacting in a manner such that their combined computing power is equivalent to that of a Turing machine [30]. These two grammars define the set of type domains and the set of function definitions operating on those domains. Note that while the term "domain" is used in a type-theoretic context, the notion can be scaled up to a much larger context as in domain of "objects." These grammars may be defined in the context 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. Each of these terms are defined below.

5.1 Types

The type declarations of a TLG program define the domains of the functions and allow strong typing of identifiers used in the function definitions. The function domains of TLG may be formally structured as linear data structures such as lists, sets, bags, or singleton data objects, or be configured as tree-structured data objects. The standard structured data types of product domain and sum domain may be treated as special cases of these.

Domain declarations have the following form:

Identifier-1, Identifier-2, ..., Identifier-m ::
`data-object-1; data-object-2; ...; data-object-n.`

where each `data-object-i` is a combination of domain identifiers, singleton data objects, and lists of data objects, which taken together as a union form the type of *Identifier-1*, *Identifier-2*, ..., *Identifier-m*. Note that if n=1, then the domain is a true singleton data object, whereas if n>1, then the domain is a set of the n objects. Syntactically, domain identifiers are capitalized, with underscores or additional capitalizations of successive words for readability (e.g., *IntegerList*, *Symbol_Table*, etc.), and singleton data objects are finite lists of natural language words written entirely in lower case letters (e.g., `sorted list`). For improved readability, the domain identifiers are represented in italics and data objects are represented in the typewriter font.

A list, set or bag structure is denoted by a regular expression or by following a domain identifier with the suffix *List*, *Set*, or *Bag*, respectively. Following conventional regular set notation, * implies a list of zero or more elements while + denotes a list of one or more elements. As in any programming language, readability is promoted through the use of appropriate names for identifiers. Furthermore, there exists a predefined environment of primitive types, defining such domains as *Integer*, *Boolean*, *Character*, *String*, etc., in the obvious ways. The main difference between list structures and tree structured domains in terms of their declaration is whether the defining domain identifier declaration is recursive or not. Recursive domains are more powerful in that they allow "context-free" data types to be defined, such as expression strings with balanced parentheses.

5.2 Functions

Function definitions comprise the operational part of a TLG specification. Their syntax allows for the semantics of the function to be expressed using a structured form of natural language. Function definitions take the forms:

function signature.
function signature : function-call-1, function-call-2, ..., function-call-n.

where n≥1. Function signatures are a combination of natural language words and domain identifiers. For improved readability, we will use boldface type to represent the function keywords. Domain identifiers in the context of a function typically correspond to variables in a conventional logic program. As in logic programs, some of these variables will typically be input variables and some will be output variables, whose values are instantiated at the conclusion of the function call. Therefore, functions usually return values through the output variables rather than directly, in which case the direct return value is considered as a Boolean `true` or `false`. `true` means that control may pass to the next function call, while `false` means the rule has failed and an alternative rule should be tried if possible. Alternative rules have the same format as that given above. If multiple function rules have the same signature, then the multiple left hand sides may be combined with a **;** separator, as in:

function signature :
function-call-11, **function-call-12**, ..., **function-call-1j**;
function-call-21, **function-call-22**, ..., **function-call-2k**;
...
function-call-n1, **function-call-n2**, ..., **function-call-nm**.

where there are n alternatives, each having a varying number of function calls. Besides Boolean values, functions may return regular values, usually the result of arithmetic calculations. In this case, only the last function call in a series should return such a value, i. e., not **function-call-1**, **function-call-2**, ..., **function-call-(n-1)**.

An important aspect about TLG is that the functions may be written at a very high level of abstraction (e.g. **compute the total mass and total cost**) or embedded into a domain definition as in traditional object-oriented programs (e.g. **compute the** *TotalMass* **and** *TotalCost* of *This Part* **by computing the** *TotalMass* **and** *TotalCost* **of its** *Subparts*, which might be embedded as a method in a *Part* class). The use of natural language in the function may be regarded as a form of infix notation for functions, in contrast with the customary prefix forms of most other programming languages. It is similar to multi-argument message selectors in Smalltalk but provides even greater flexibility, including the presence of logical variables, denoted by the use of domain names (capitalized). This notation provides a highly readable way of writing what is to be done and is wide-spectrum in the sense that "what is to be done" may be expressed at multiple levels. The functions typically return a Boolean value as the main operation is to instantiate the logical variables as in Prolog, but simple function values such as arithmetic expressions may also be computed.

To explain the operational semantics of Two-Level Grammar function rules, note that each function call on the right hand side of a function definition should correspond to a function signature defined within the scope of the TLG program or be a special operation such as a Boolean comparison, assignment statement, or if-then-else statement. The most important aspect of function definitions is that every domain identifier with the same name is instantiated to the same value, as in Prolog. This is called *consistent substitution*. If variables have the same root name but are numbered, then the numbers are used to distinguish between variables. A numbered variable *V1* will then be different from a variable *V2* and the two can have different values. However, they will be of the same type, namely type *V*. Note that once a variable has been assigned a value, it may not be reassigned, unless it is an instance variable of a class, and even in this case, it would not be usual to do so in the same function. Each function definition may therefore be thought of as a set of logical rules. Also as in Prolog, the function calls are executed in the order given in the function definition. Functions may be recursive with the expected operational behavior.

Besides defined functions, TLG supports the usual arithmetic and Boolean operations. For lists, list comprehensions are also supported as are iterators over the list. The syntax of a list comprehension is **list all** *Element* **from** *ElementList1* **such that** *Element* **condition giving** *ElementList2*. This returns a list, *ElementList2*, of all *Element* values in *ElementList* satisfying the given condition. The syntax of an iterator is **select** *Element* **from** *ElementList* **with** *Element* **condition**. This returns the first *Element* from *ElementList* which satisfies the condition.

5.3 Classes

In order to support object-orientation, TLG domain declarations and associated functions may be structured into a class hierarchy supporting multiple inheritance. The syntax of TLG class definitions is:

class *Identifier-1* [**extends** *Identifier-2*, *Identifier-3*, ... *Identifier-n*].
    instance variable and method declarations
end class [*Identifier-1*].

In the above syntax, square brackets are used to indicate the construct is optional. *Identifier-1* is declared to be a class which inherits from classes *Identifier-2*, *Identifier-3*, ..., *Identifier-n*. Note that the **extends** clause is optional so a class need not inherit from any other class. The instance variables comprising the class definition are declared using the domain declarations described earlier. In general, the scope of these domain declarations is limited to the class in which they are defined, while the methods, corresponding to TLG function definitions, have scope anywhere an object of the given class is referred to. These notions of scoping correspond to *private* and *public* access respectively in object-oriented languages such as C++ and Java, and either scope may be declared explicitly or the scope may be made *protected*. Methods are called by writing a sentence or phrase containing the object. The result of the method call is to instantiate the logical variables occurring in the method definition.

In any class for every instance variable of simple type there are **get** and **set** methods to access or modify that variable.

TLG class declarations serve to encapsulate the TLG domain declarations and function definitions. The class hierarchy which is resident in TLG is a small forest of built-in classes, such as integers, lists, etc. The "main" program is nothing more than a set of object declarations using the existing class identifiers as domain names and a "query" of the appropriate methods.

5.4 Example

As an example of a TLG specification, consider the following translation scheme for producing three address code [1] from simple arithmetic expressions.

class *CodeGenerator*.
    Expression :: *Term* {*AddingOperator Term*}*.
    AddingOperator :: `+`; `-`.
    Term :: *Factor* {*MultiplyingOperator Factor*}*.
    MultiplyingOperator :: `*`; `/`.
    Factor :: `(` *Expression* `)`; *Identifier*; *Float*; *Integer*.
    ExpressionIdentifier, TermIdentifier, FactorIdentifier, Identifier :: *String*.
    ExpressionType, TermType, FactorType, Type :: `float`; `integer`; `undefined`.

    three address code for *Expression AddingOperator Term* **is** *Identifier* **type** *Type* :
        three address code for *Expression* **is** *ExpressionIdentifier* **type** *ExpressionType*,
        three address code for *Term* **is** *TermIdentifier* **type** *TermType*,
        common type of *ExpressionType* **and** *TermType* **is** *Type*,
        type convert *ExpressionIdentifier* **type** *ExpressionType* **into** *Identifier1* **type** *Type*,
        type convert *TermIdentifier* **type** *TermType* **into** *Identifier2* **type** *Type*,
        ThreeAddressCode **generate temporary** *Identifier* **:=** *Identifier1 AddingOperator Identifier2*.

    three address code for *Term MultiplyingOperator Factor* **is** *Identifier type Type* :
        ... similar to above ...

    three address code for `(` *Expression* `)` **is** *Identifier* **type** *Type* :
        three address code for *Expression* **is** *Identifier* **type** *Type*.

    three address code for *Identifier* **is** *Identifier* **type** *Type* :

SymbolTable **lookup** *Identifier* **giving** *Type*,
Type != `undefined`.

three address code for *Float* **is** *Float* **type** `float`.

three address code for *Integer* **is** *Integer* **type** `integer`.

....

end class.

For simplicity only two types, **float** and **integer**, are assumed. There is also a *SymbolTable* class assumed with standard operations such as looking up an identifier to obtain its type, and a *ThreeAddressCode* class assumed with an operation to generate a three-address code instruction in the code array, possibly including an assignment to a temporary variable. Rules to check type compatibility and perform type conversions are also present but not shown here. Error checking is not explicitly indicated but would occur through failure of any rule, e.g., a syntactically ill-formed expression would not match any of the **three address code** rules, an identifier not declared would cause the identifier rule to fail, and any errors in typing would cause the type checking rules to fail.

These rules would be queried as follows:

CodeGenerator **three address code for** `a * (b + 1)` **is** *Id* **type** *Type*

This creates a code string of:

```
t1 := b + 1
t2 := a * t1
```

and returns `t2` for *Id* and `integer` for *Type*, respectively (assuming that `a` and `b` are stored in the symbol table as type `integer` variables).

This example illustrates that TLG may be used to provide for attribute evaluation and transformation, syntax and semantics processing of languages, parsing, and code generation. All of these are required to use TLG as a specification language for generative rules.

5.5 Implementation

Two-Level Grammar is implemented as part of a specification development environment which facilitates the construction of TLG specifications from natural language, and then translates TLG specifications into executable code. The natural language requirements are translated into TLG through Contextual Natural Language Processing (CNLP) [19] which constructs a knowledge representation of the requirements which may then be expressed using TLG. The TLG is then translated into VDM++ [12], the object-oriented extension of the Vienna Development Method (VDM) specification language [16]. The IFAD VDM Toolbox™ [13] may then be used to generate code in an object-oriented programming language such as Java or C++.

6 A Case Study

6.1 Client/Server Distributed System

In order to highlight the significance of UA, along with functions of each of its constituents, a simple example of an account management system is described below. This system consists of two categories of components – `AccountServer` and `AccountClient`. There are two instances of `AccountServer` and one instance of `AccountClient`. The server components are heterogeneous – `javaAccountServer` adheres to the Java-RMI model; while `corbaAccountServer` is developed using the CORBA model. The client, `javaAccountClient` is developed by using the Java-RMI model and is implemented as an applet. The goal is to assemble an account management system from these available components. The partial UMM descriptions of these components are presented below.

6.2 Component Description in UMM

```
javaAccountServer
-----------------
Informal Description: Provides an account management service. Supports three
functions: javaDeposit(), javaWithdraw() and javaBalance().

1. Computational Attributes:
    a) Inherent Attributes:
        a.1 id: intrepid.cs.iupui.edu/jServer

    b) Functional Attributes:
        b.1 Acts as an account server
        b.2 Algorithm: simple addition/subtraction
        b.3 Complexity: O(1)
        b.4 Syntactic Contract:
            void javaDeposit(float ip);
            void javaWithdraw(float ip) throws overDrawException;
            float javaBalance();
b.5 Technology: Java-RMI
              .......

2. Cooperation Attributes:
    2.1) Pre-processing Collaborators: AccountClient

3. Auxiliary Attributes:
              .......
4. QoS Metrics:
    Availability: 90%
    End-to-End Delay < 10 ms

corbaAccountServer
------------------
Informal Description: Provides an account management service. Supports three
functions: corbaDeposit(), corbaWithdraw() and corbaBalance().

1. Computational Attributes:
    a) Inherent Attributes:
        a.1 id: jovis.cs.iupui.edu/coServer

    b) Functional Attributes:
        b.1 Acts as an account server
        b.2 Algorithm: simple addition/subtraction
        b.3 Complexity: O(1)
        b.4 Syntactic Contract:
            void corbaDeposit(float ip);
            void corbaWithdraw(float ip) throws overDrawException;
            float corbaBalance();
        b.5 Technology: Java-CORBA
              .......

2. Cooperation Attributes:
    2.1) Pre-processing Collaborators: AccountClient

3. Auxiliary Attributes:
              .......
```

```
4. QoS Metrics:
    Availability: 95%
    End-to-End Delay < 10 ms

javaAccountClient
--------------
Informal Description: Requests account services from an appropriate server and
interacts with the user -- implemented as a web-based applet. Supports
functions: depositMoney(), withdrawMoney() and checkBalance().

1. Computational Attributes:
    a) Inherent Attributes:
        a.1 id: galileo.cs.iupui.edu/aClient

    b) Functional Attributes:
        b.1 accepts user queries and presents the results using a GUI
        b.2 Algorithm: Java Foundation Classes (JFC)
        b.3 Complexity: O(1)
        b.4 Syntactic Contract
            void depositMoney(float ip);
            void withdrawMoney(float ip);
            float checkBalance();
        b.5 Technology: Java Applet
        .......

2. Cooperation Attributes:
    2.1) Post-processing Collaborators: AccountServer

3. Auxiliary Attributes:
             .......

4. QoS Metrics:
    Availability: 80%
    End-to-End Delay < 20 ms
```

6.3 Query and Description of Target Code Architecture

Queries are stated in a structured form of natural language and then processed into TLG. 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 follows the standards outlined earlier. A sample query for the above example can be informally stated as: *Create an account management system that has: availability = 50% and end-to-end delay < 50 ms.* This query specifies satisfaction of one static and one dynamic QoS parameter. From the query and the available knowledge in the GDM associated with the account management systems, a formal specification of the desired system will be formulated for a headhunter in the UMM. In response, the headhunter will discover the following choices:

1. Java-Java System
   (a) `javaAccountClient` – availability = 80%, End-to-End delay < 20ms, Java Applet Technology
   (b) `javaAccountServer` – availability = 90%, End-to-End delay < 10ms, Java-RMI technology
   (c) Infrastructure Needed – JVM and Appletviewer
2. Java-CORBA System
   (a) `javaAccountClient` – availability = 80%, End-to-End delay < 20ms, Java Applet Technology
   (b) `corbaAccountServer` – availability = 95%, End-to-End delay < 10ms, Java-RMI technology
   (c) Infrastructure Needed – JVM, Appletviewer, ORB, Java-CORBA bridge

6.4 TLG Specification

Two-Level Grammar 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.

6.4.1 UMM

The basic TLG specification of the UMM specification template is given below. Some parts are omitted for brevity. Any system described using UMM will be typed according to these declarations.

UMM :: *ComponentName InformalDescription FunctionList ComputationalAttributes*
CooperationAttributes AuxiliaryAttributes QoSMetricList.
ComponentName, *InformalDescription*, *Function* :: *String.*
ComputationalAttributes :: *InherentAttributes FunctionalAttributes.*
InherentAttributes :: *Id Version DateDeployed.*
Id :: *String.*
Version :: *Float.*
DateDeployed :: *Date.*
FunctionalAttributes :: *TaskDescription AlgorithmAndComplexity SyntacticConstruct Technology.*
TaskDescription :: *String.*
AlgorithmAndComplexity :: ....
SyntacticConstruct :: *FunctionSignatureList.*
FunctionSignature :: ....
Technology :: `corba`; `java applet`; `java rmi`; ....
CooperationAttributes :: *PreprocessingCollaboratorList PostprocessingCollaboratorList.*
PreprocessingCollaborator :: *String.*
PostprocessingCollaborator :: *String.*
AuxiliaryAttributes :: ....
QoSMetric :: *Throughput*; *Capacity*; *EndToEndDelay*; *ParallelismConstraints*; *Availability*; ....
Throughput :: *Float.*
Capacity :: *Integer.*
EndToEndDelay :: *Integer* `ms`.
ParallelismConstraints :: `synchronous`; `asynchronous`.
Availability :: *Float*; *Integer* `%`.
....

It can be seen that the previous UMM examples may be parsed using the above TLG. This parsing provides a structure to the UMM that can be processed by TLG functions. These functions include generative rules for construction of the wrapper/glue code and the event grammar instrumentation to assure the QoS of the accounting system. The GDM for account management systems will be described according to this template, including both generation rules and QoS parameter processing.

6.4.2 Generation Rules

A sampling of TLG rules which may be used to generate the appropriate glue/wrapper code to connect the components of the accounting system are presented below. These rules are based on selecting from the GDM of the accounting systems the appropriate system model for this two-component DCS.

ClientUMM, ServerUMM :: *UMM.*
ClientOperations, ServerOperations :: {*Interface*}*.
generate system from *ClientUMM* **and** *ServerUMM* **giving** *JavaCode* :
get operations from *ClientUMM* **as** *ClientOperations*,
get operations from *ServerUMM* **as** *ServerOperations*,

map *ClientOperations* **into** *ServerOperations* **as** *OperationMapping*,
get component model from *ServerUMM* **giving** *ComponentModel*,
generate java code for *OperationMapping* **using** *ComponentModel* **giving** *Code*.

This rule generates Java code for two UMM models representing a client and server, respectively. For this example, the *ClientUMM* would be the UMM specification of `javaAccountClient` presented previously and the *ServerUMM* would be the UMM specification of `javaAccountServer` or `corbaAccountServer`. 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 generated code will be in Java since the client code is in Java and must seamlessly interface with it. If the client is in C++ or other language, similar rules will be defined and many rules will be language independent.

The actual mapping to be defined will be based upon a natural language analysis of the names of operations. The closer the names match, the more easily the system can establish the correct mapping. This depends upon both the care and style with which the user has written the interface method names and so may vary widely. For this example, it can be seen that the correspondence between names, while not exact, is relatively close.

The next rule describes the specifics of generating CORBA code in Java to implement the mapping that arises by combining the `javaAccountClient` with the `corbaAccountServer`.

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

This rule generates the class structure required by the Java implementation, which 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" concatenated [5] with the name of the server class, i.e., *JavaCorbaAccountServer*. The name of the CORBA object is generated in a similar way.

The rule below describes the mechanism for generating the individual methods in *JavaCorbaAccountServer*. For simplicity, only the case where the class is to contain a single method is shown. Multiple methods would be handled in a similar manner.

generate java code for *OperationName1 ArgumentList1 ReturnType* **maps to**
OperationName2 ArgumentList2 ReturnType **giving**
`public` *JavaReturnType OperationName1* `(` *JavaArgumentListDefinition* `) {`

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

OperationCall
`}` :
java type of *ReturnType* **is** *JavaReturnType*,
list all *Argument* **from** *ArgumentList1*
mapped to *JavaArgument* **by function java argument of** *Argument* **is** *JavaArgument*
giving *JavaArgumentList*,
separate *JavaArgumentList* **by , giving** *JavaArgumentListDefinition*,
generate java code for *OperationName2 ArgumentList1 ReturnType* **giving** *OperationCall*.
...

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 and generate the appropriate method call. The former is accomplished by using a TLG list comprehension to map the arguments in *ArgumentList1* into corresponding Java arguments represented by *JavaArgumentList*. 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 ReturnType* **giving**
`return` *CorbaObjectName* `.` *OperationName* `(` *IdentifierListInCall* `);` :
list all *Argument* **from** *ArgumentList*
mapped to *Identifier* **by function argument id of** *Argument* **is** *Identifier*
giving *IdentifierList*,
separate *IdentifierList* **by , giving** *IdentifierListInCall*.

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** *Identifier*). Likewise, the abstract syntax representation must be made concrete by comma separators.

For the example UMM specification, the following code for the `depositMoney` function would be produced.

```
public void depositMoney (float ip) {
  return objectCorbaAccountServer . deposit (ip);
}
```

6.5 QoS

Each component has two QoS parameters - 1) static - run-time availability (e.g. 90% and 95% respectively) and 2) dynamic - end-to-end delay measured in milliseconds. The desired QoS of the assembled system includes both of these parameters as well. For this reason the GDM will contain a rule for the static parameter that will multiply the various availability parameters (e.g. obtaining 85.5% availability for the assembled system in this case), assuming component availability is independent.

For the dynamic parameter, the generator will provide the necessary instrumentation for taking the clock and calculating the end-to-end delay at run-time. The knowledge about metrics for the QoS parameter 'end-to-end delay' is represented in terms of Duration attribute for events of the type method-call, and the generic computation over the event trace that takes the clock and sums up those durations yielding a measured end-to-end delay time for the accounting system.

One of the two example systems, mentioned in the section 6.3, will be implemented with the code for carrying out event trace computations according to user supplied test cases. These test cases will be executed to verify that the accounting system satisfies the QoS specified in the query of the section 6.3. If the system is not verified, it is discarded. This verification process is carried out for each of the generated accounting system (two in the above example). Then the one with the best QoS is chosen, in the above example this the `corbaAccountServer` and `javaAccountClient` combination.

7 Conclusion

This paper has presented a framework that allows an interoperation of heterogeneous and distributed software components. This framework incorporates the following key concepts: a) a meta-component model, b) integration of QoS at the individual component and distributed system levels, c) validation and assurance of QoS, based on the concept of event grammars, d) formal specification, based on two-level grammar, of each component and associated queries for integrating a distributed system, and e) generative rules, along with their formal specifications, for assembling an ensemble of components out of available choices. The software solutions for future DCS will require either automatic or semi-automatic integration of software components, while abiding by the QoS constraints advertised by each component and the system of components. The result of using UMM and the associated tools is a semi-automatic construction of a distributed system. Although a simple case study is provided in this paper, the principles of the proposed approach are general enough to be applied to any larger application examples.

Acknowledgments. The material presented in this paper is based upon work supported by, or in part by, a) the U. S. Office of Naval Research under award number N00014-01-1-0746, b) the U. S. Army Research Laboratory and the U. S. Army Research Office under contract/grant number 40473-MA, and c) the U. S. Army Research Laboratory and the U. S. Army Research Office under contract/grant number DAAD19-00-1-0350.

References

[1] Aho, A. V. and Sethi, R. and Ullman, J. D. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] Auguston, M. Program Behavior Model Based on Event Grammar and its Application for Debugging Automation. In *Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging*, 1995.

[3] Auguston, M. and Gates, A. and Lujan, M. Defining a Program Behavior Model for Dynamic Analyzers. In *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering, SEKE'97*, pages 257–262, 1997.

[4] Batory, D. and Chen, G. and Robertson, E. and Wang, T. Design Wizards and Visual Programming Environments for GenVoca Generators. *IEEE Transactions on Software Engineering*, pages 441–452, 2000.

[5] BBN Corporation. *Quality Objects (Quo) URL:-http://www.dist-systems.bbn.com/tech/QuO/*, 2001.

[6] Bryant, B. R. Object-Oriented Natural Language Requirements Specification. In *Proceedings of ACSC 2000, the 23rd Australasian Computer Science Conference, January 31-February 4, 2000, Canberra, Australia*, pages 24–30, January 2000.

[7] California Institute of Technology. *Caltech Infospheres On-line Documentation, URL:- http://www.infospheres.caltech.edu/*, 1998.

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

[9] Fox, G. The Document Object Model Universal Access Other Objects CORBA XML Jini JavaScript etc. *http://www.npac.syr.edu/users/gcf/msrcobjectsapril99*, 1999.

[10] Gamma, E., Helm R., Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley Publication Company, 1995.

[11] Globus Project. *Globus Website, URL:- http://www.globus.org/*, 2000.

[12] IFAD. The IFAD VDM++ Language. Technical report, IFAD, 1999.

[13] IFAD. The VDM++ Toolbox User Manual. Technical report, IFAD, 2000.

[14] Israel, B. and Kaiser, G. Coordinating Distributed Components Over the Internet. *IEEE Internet Computing*, pages 83–86, 2(2), 1998.

[15] Jurafsky, D. and Martin, J. H. *Speech and Language Processing.* Prentice Hall, 2000.

[16] Larsen, P. G., et al. Information Technology - Programming Languages, Their Environments and System Software Interfaces - Vienna Development Method - Specification Language - Part I: Base Language. Report, International Standard ISO/IEC 13817-1, December 1996.

[17] Luqi, Berzins, V., Ge, J., Shing, M., Auguston, M., Bryant, B. R. and Kin, B. K. DCAPS - Architecture for Distributed Computer Aided Prototyping System. In *Proceedings of RSP 2001, the 12th International Workshop on Rapid System Prototyping*, 2001.

[18] Manola, F. Technologies for a Web Object Model. *IEEE Internet Computing*, 3(1):38–47, January-February 1999.

[19] McCarthy, J. Notes on Formalizing Context. Technical report, Computer Science Department, Stanford University, Stanford, CA, 1993.

[20] Michigan State University. *RAPIDware: Component-Based Development of Adaptable and Dependable Middleware URL:-http://www.cse.msu.edu/rapidware/*, 2001.

[21] Microsoft Corporation. *DCOM Specifications, URL:- http://www.microsoft.com/oledev/olecom*, 1998.

[22] Object Management Group. CORBA Components. Technical report, Object Management Group TC Document orbos/99-02-05, March 1999.

[23] Object Management Group. Model Driven Architecture: A Technical Perspective. Technical report, Object Management Group Document No. ab/2001-02-01/04, February 2001.

[24] Orfali R, and Harkey, D. *Client/Server Programming with JAVA and CORBA.* John Wiley & Sons, Inc., 1997.

[25] Peng, S. and Raje, R. A Comprehensive Framework on Quality of Service for Software Components. Technical report, CIS Department, IUPUI (to become a technical report soon), May 2001.

[26] Raje, R. UMM: Unified Meta-object Model for Open Distributed Systems. In *Proceedings of the fourth IEEE International Conference on Algorithms and Architecture for Parallel Processing (ICA3PP'2000)*, 2000.

[27] Raje, R., Auguston, M., Bryant, B., Olson, A. and Burt, C. A Unified Approach for the Integration of Distributed Heterogeneous Software Components. In *Proceedings of the 2001 Monterey Workshop (To Appear)*, 2001.

[28] Rogerson, D. *Inside COM.* Microsoft Press, 1996.

[29] Siegel, J. *CORBA Fundamentals and Programming.* John Wiley & Sons, Inc., 1996.

[30] Sintzoff, M. Existence of van Wijngaarden's Syntax for Every Recursively Enumerable Set. *Ann. Soc. Sci. Bruxelles*, 2:115–118, 1967.

[31] University of Virginia. *Legion Project, URL:- http://www.cs.virginia.edu/ legion*, 1999.

[32] van Wijngaarden, A. Orthogonal Design and Description of a Formal Language. Technical report, Mathematisch Centrum, Amsterdam, 1965.

[33] van Wijngaarden, A. Revised Report on the Algorithmic Language ALGOL 68. *Acta Informatica*, 5:1–236, 1974.

[34] Zinky, J. A., Bakken, D. E., and Schantz, R. Overview of Quality of Service for Distributed Objects. In *Proceedings of the Fifth IEEE Dual Use Conference*, 1995.