

A Unified Approach for the Integration of Distributed Heterogeneous Software Components¹

Rajeev R. Raje^{2 3} Mikhail Auguston^{4 5} Barrett R. Bryant^{4 6} Andrew M. Olson² Carol Burt⁷

Abstract

Distributed systems are omnipresent these days. Creating efficient and robust software for such systems is a highly complex task. One possible approach to developing distributed software is based on the integration of heterogeneous software components that are scattered across many machines. In this paper, a comprehensive framework that will allow a seamless integration of distributed heterogeneous software components is proposed. This framework involves: a) a meta-model for components and associated hierarchical setup for indicating the contracts and constraints of the components, b) an automatic generation of glues and wrappers, based on a designer's specifications, for achieving interoperability, c) a formal mechanism for precisely describing the meta-model, and d) a formalization of quality of service (QoS) offered by each component and an ensemble of components. A case study from the domain of distributed information filtering is described in the context of this framework.

Keywords: Distributed systems, Formal methods, Glue and Wrapper technology, Quality of Service

1 Introduction

The rapid advances in the processor and networking technologies have changed the computing paradigm from a centralized to a distributed one. This change in paradigm is allowing us to develop distributed computing systems (DCS). DCS appear in many critical domains and are, typically, characterized by: a) a large number of geographically dispersed and interconnected machines, each containing a subset of the required data, b) an open architecture, c) a local autonomy over the hardware and software resources, d) a dynamic system configuration and integration, e) a time-sensitivity of the expected solution, and f) the quality of service with an appropriate notion of compensation. These characteristics make the software design of DCS an extremely difficult task.

One promising approach to the software design of DCS is based on the principles of distributed component computing. Under this paradigm DCS are created by integrating geographically scattered heterogeneous software components. These components constantly discover one another, offer/utilize services, and negotiate the cost and the quality of the services. Such a view provides a scalable solution and hides the underlying heterogeneity.

Various distributed component models, each with strengths and weaknesses, are prevalent and widely used. However, almost a majority of these models have been designed for 'closed' systems, i.e., systems, although distributed in nature, are developed and deployed in a confined setup. In contrast, a direct consequence of the heterogeneity, local autonomy and the open architecture is that the software realization of DCS requires combining components that adhere to different distributed models. This in turn increases the complexity of the design process of DCS. Hence, a comprehensive framework, that provides a seamless access to underlying components and aids in the design of DCS, is needed.

In this paper, one such framework is described. This framework consists of: a) a meta-model for components and associated hierarchical setup for indicating the contracts and constraints of the components, b) an automatic generation of glue and wrappers, based on a designer's specifications, for achieving interoperability, c) a formal mechanism for precisely describing the meta-model, and d) a formalization of the notion of quality of service offered by each component and an ensemble of components. The paper also presents a case study that shows the application of the framework to a specific problem domain.

The rest of the paper is organized as follows. The next section contains a detailed discussion about the meta-model. As an application of the meta model, a case study from the domain of distributed information filtering is presented in the Section 3. Section 4 deals with the formal specification of the meta model, the automated system integration, and evaluation of the approach. Finally, we conclude in Section 5.

¹This material is based upon work supported by, or in part by, the U. S. Office of Naval Research under award number N00014-01-1-0746.

²Department 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

³This material is based upon work supported by, or in part by, the National Science Foundation Digital Libraries Phase II grant.

⁴Computer Science Department, Naval Postgraduate School, 833 Dyer Rd., SP 517, Monterey, CA 93943, USA, {auguston, bryant}@cs.nps.navy.mil, +1 831 656 2509/2726

⁵This 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 number 40473-MA. On leave from Computer Science Department, New Mexico State University, USA.

⁶This 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 number DAAD19-00-1-0350. On leave from Department of Computer and Information Sciences, University at Alabama at Birmingham, USA.

⁷2AB, Inc., 1700 Highway 31, Calera, AL 35040, USA, cburt@2ab.com, +1 205 621 7455

2 Component Models and a Meta-model

Many models and projects for the software realization of DCS have been proposed by academia and industry. A few prominent ones are: JavaTM Remote Method Invocation (RMI) [16], Common Object Request Broker Architecture (CORBATM) [16, 20], Distributed Component Object Model (DCOMTM) [11, 16], Web-component model/DOM [10], Pragmatic component web [5], Hadas [6], Infospheres [4], Legion [22], and Globus [21]. Each of these models/projects has strengths and weaknesses. Some of these are language-centric and only assume a uniform way of the world (Java); while the others allow a limited interoperability (CORBA – allowing implementations in different languages). Some of these are general-purpose, i.e., not concentrating on any particular application domain (DCOM), while others are specifically tailored to high-performance computing applications (Legion). However, almost all of these models/projects do not assume the presence of other models. Thus, the interoperability which they provide is limited mainly to the underlying hardware platform, operating system and/or implementational languages. Also, there are hardly any models which emphasize the notion of quality of service offered by the components. Projects, such as Agent TCL [8], etc., based on the principles of intelligent agents have imbibed the notion of the quality of service and related compensation. However, the agents are at a higher level of abstraction than components and many of the agent projects/frameworks use one or the other existing distributed-component models at the low-level.

2.1 Why a Meta-model?

Given the above mentioned plethora of component-based models and also noting the fact that components, by their 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 questions 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 DCS by combining components then the quality of service 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 rapidly create prototypes 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 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 offered by each component and an amalgamation of components.

2.2 Unified Meta-component Model (UMM)

In [17] we have proposed a unified meta-component model (UMM) for global-scale systems. 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 (CCMTM) [13] and Java Enterprise Edition component models (J2EETM) 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 (COMTM) [18] 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.

For enterprise component solutions, this is an area where significant standards work is now focused. The OMG Meta Object Facility (MOFTM) [14] provides a common meta-model that allows the interchange of models between tools as well as the expression of models in XMITM (an MOF compliant XMLTM (eXtended Markup Language)) [12]. This work allows the generation of interfaces from Unified Modeling Language (UML) [19] models, however, a careful analysis of the resulting interface specifications makes it clear that distribution is not a key factor in the algorithms used. For example, quality of service requirements for performance, scalability and/or security would dictate the use of iterators, the factoring of interfaces to separate “query” and “administrative” operations, and the use of structures and/or objects passed by value. The current standards in this tend to focus on data access with accessors and mutators and relationship transversal. This is acceptable in a single machine environment, but unacceptable for highly distributed communications and collaborations. The recent shift in focus for the Object Management Group to “Model Driven Architecture” (MDATM) [15] is a recognition that to create mechanized software for the collaboration and bridging of component architectures will require standardization

of 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.

The following sections describe the various aspects of UMM in detail.

2.2.1 Component

In UMM, components are autonomous entities, whose implementations are non-uniform, i.e., each component adheres to some distributed-component model and there is no notion of either a centralized controller or a unified implementational framework. Each component has a state, an identity and a behavior. Thus, all components have well-defined interfaces and private implementations. In addition, each component in UMM has three aspects: 1) a computational aspect, 2) a cooperative aspect, and 3) an auxiliary aspect.

Computational Aspect

The computational aspect reflects the task(s) carried out by each component. It in turn depends upon: a) the objective(s) of the task, b) the techniques used to achieve these objectives, and c) the precise specification of the functionality offered by the component. In DCS, components must be able to ‘understand’ the functionality of other components. Thus, each component in UMM supports the concept of introspection, by which it will precisely describe its service to other inquiring components. There are various alternatives for a component to indicate its computation – ranging from simple text to formal descriptions. Both these extremes have advantages and drawbacks. 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 functional part is formal and indicates precisely the computation, its associated contracts and the level(s) of service offered by the component. Multi-level contracts for components have been proposed by [2], classifying the contracts into four levels – syntactic, behavioral, concurrency and quality of service (QoS). UMM integrates this multi-level contract concept into the functional part of the computational aspect. As stated earlier, in DCS each component will be offering a service and hence, the level related to the QoS is especially critical in UMM. The QoS depends upon many factors such as, the algorithm used, the execution model, resources required, time, precision and classes of the results obtained. UMM makes an attempt at quantifying the QoS by creating a vocabulary and providing multiple levels of quality, which could be negotiated by the components involved in an interaction. The functional part will also be specified by the creator of the component.

Cooperative Aspect

In UMM, components are always in the process of cooperating with each other. This cooperation may be task-based or greed-based. The cooperative aspect depends on many factors: detection of other components, cost of service, inter-component negotiations, aggregations, duration, mode, and quality. Informally, the cooperative aspect of a component may contain: 1) Expected collaborators – other components that can potentially cooperate with this component, 2) Pre-processing collaborators – other components on which this component depends upon, and 3) Post-processing collaborators – other components that may depend on this component.

Auxiliary Aspect

In addition to computation and cooperation, mobility, security, and fault tolerance are necessary features of DCS. The auxiliary aspect of a component will address these features. In UMM, each component can be potentially mobile. The mobility of the component will be shown as a ‘mobility attribute’ (a notion similar to the inherent attribute). If a component is mobile, then the mobility attribute will contain the necessary information, such as its implementation details and required execution environment. Similarly, security in DCS is a critical issue. The security attribute of a component will contain the necessary information about its security features. As DCS are prone to frequent failures, full and partial, fault tolerance is critical in these systems. Similar to mobility and security, each component contains fault-tolerant attributes in its auxiliary aspect.

2.2.2 Service and Service Guarantees

The concept of a service is the second part of the UMM. A service could be an intensive computational effort or an access to underlying resources. In DCS, it is natural to have several choices for obtaining a specific service. Thus, each component, in addition to indicating its functionality, must be able to specify the cost and quality of the service offered.

The nature of the service offered by each component is dependent upon the computation performed by that component. In addition to the algorithm used, expected computational effort and resources required, the cost of each service will be decided by the motivation of the owner and the dynamics of supply and demand. In a dynamic environment costs must always be accompanied by the duration for which the costs are valid. As the system dynamics undergo constant changes, the methodologies used to fix the cost of a service will evolve as time progresses, thereby creating a need to indicate the time sensitiveness of the cost. The quality of service is an indication given by an component, on behalf of its owner, about

its confidence to carry out the required services in spite of the constantly changing execution environment and a possibility of partial failures. The techniques used to determine the cost, the time-validity and the quality of a service will depend upon the tasks carried out by the component and the objectives of its owner and will involve principles of distributed decision making.

There are many parameters that a component can use to indicate its quality of service. A few examples are: i) Throughput – number of methods executed per second and classification of methods based on their read/write behaviors, ii) Parallelism constraints – synchronous or asynchronous, iii) Priority, iv) Latency or End-to-End Delay – turn-around time for an invocation, v) Capacity – how many concurrent requests a given component can handle, vi) Availability – indication of the reliability of a component, vii) Ordering constraints – can invocations (asynchronous) be executed out of order by a component, viii) Quality of the result returned – does the component provide a classification or ranking of the result, and ix) Resources available – how many resources (hardware/data) are accessible to the component under consideration and what are the types of resources.

When a component uses certain metrics to indicate its QoS (either all the mentioned criteria or a sub/super set of them), three interesting issues need to be addressed: a) how does the component developer decide these parameters?, b) how does the developer guarantee the advertised QoS during the execution?, and c) when components are collected together as a solution for specific DCS, what happens to the QoS of the combination and how does the combined QoS meet the quality requirements of DCS?

The parameters to be used to describe the QoS of a component are highly context (application) dependent. The proposed approach is to create lists of QoS metrics for common application domains. A few examples of such domains are: scientific computing, multi-media applications, information filtering, and databases. Once such lists are created, they would be used as a template by the component developers while advertising the QoS of their components.

QoS of Components

The issue of guaranteeing a particular QoS, for a component, in an ever changing dynamic DCS is extremely critical; mainly because of external (e.g., policy matters related to resources) and internal (e.g., changes in algorithms) factors that affect a life cycle of a component. In addition, as the software realization of DCS is based on an amalgamation of heterogeneous components, a proper guarantee of a QoS offered by a component effectively decides the QoS of the entire DCS. The quality metrics are expected to vary from one application domain to another and which metrics to select would depend on the intentions of the component developer and the functionality offered by that component. A few examples of such QoS metrics are already mentioned in the previous section. Irrespective of the metrics selected, there is a need for a well-defined mechanism that will assist the developer to achieve the necessary QoS when that component is deployed. Just like any software development process, the process of guaranteeing a certain QoS, as offered by a component, will be an incremental and iterative one, as will be discussed later.

QoS of an Integrated System

In addition to the QoS of individual components, there is a need to achieve a certain QoS for the ensemble of heterogeneous components assembled for a distributed system under discussion. The QoS of such an amalgamation will be decided by the design constraints of the system under construction. However, the integral characteristics of such a system typically cannot be expressed as a function of individual components but as a property of the whole system behavior. Hence, there is a need for a formal model of system behavior, which will integrate the behaviors of each component in the ensemble along with its QoS guarantees.

The proposed approach to address the problem of QoS is as follows. First, build a precise model of systems behavior (event trace notion), provide a programming formalism to describe computations over event traces, and then apply these in order to define different kinds of QoS metrics. Constructive calculations of QoS metrics on a representative set of test cases is one of cornerstones of the proposed iterative approach to system assembly from components meeting user's query specifications.

This approach to the design of a system behavior model assumes that the run time actions performed within the system may be observed as detectable events. Each event corresponding to an action is a time interval, with beginning, end, and duration. Certain attributes could be associated with the event, e.g. program state, source code fragment, time, etc. There are two binary relations defined for the event space: inclusion (one event may be nested within another), and precedence (events may be partially ordered accordingly to the semantics of the system under consideration). Hence, when executed, a system generates an event trace - set of events structured along the relations above. This event trace actually can be considered as a formal behavior model of the system ("lightweight semantics"). This model could be presented as a set of axioms about event trace structure called event grammar [1].

For example, suppose that the entire system execution is represented by an event of type `execute-system`. It may contain events of the type `evaluate-component-A` and `evaluate-component-B`. Event grammar may contain an axiom: `execute-system: (evaluate-component-A evaluate-component-B)*` which states that `evaluate-component-A` is always followed by the `evaluate-component-B` event, and these pairs may be repeated zero or more times.

A new concept for specification and validation of target program behavior based on the ideas of event grammars and

computations over program execution traces has been developed, and assertion language mechanisms, including event patterns and aggregate operations over event traces, to specify expected behavior, to describe typical bugs, and to evaluate debugging queries to search for failures (e.g. gathering run time statistics, histories of program variables, etc.) have been created. An event grammar provides a basis for QoS metrics implementation via target program automatic instrumentation. Since the instrumentation is conditional, it does not deteriorate the efficiency of the final version generated code. This mechanism based on independent models of system behavior makes it possible to define QoS metrics as generic trace computations, so that the same metric may be applied to different versions of an assembled system (via automatic instrumentation). To facilitate use of the event grammar model for the assembled system, the event definitions should be consistent through the entire component space. The QoS metrics for components should adhere to this principle. The process proposed in Section 4.4 for assembling a distributed system from components in a distributed network offers a possible approach to achieving this.

2.2.3 Infrastructure

As local autonomy is inherent in open DCS, forcing every component developer to abide by certain rigid rules, although attractive, 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 a cooperation among heterogeneous (adhering to different models) components.

Head-hunter Components

The tasks of head-hunters are to detect the presence of new components in the search space, register their functionalities, and attempt at 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, i.e., the onus of registration is on the foreign components and not on the trader. In contrast, a headhunter is active, i.e., it discovers other components and makes an attempt to register them with itself. There are many approaches possible for the discovery of components. They range from the standard search techniques to broadcasts and multi-casts to selected machines. At a conceptual basis, UMM does not tie itself to a specific approach but during the prototype development a particular approach will be selected for the discovery process. During registration, each component will inform the head hunter about all its aspects. The head hunter will use this information during matching. A component may be registered with multiple head-hunters. Head-hunters may cooperate with each other in order to serve a large number of components. The functionality of head hunters makes it necessary for them to communicate with components belonging to any model, implying that the cooperative aspect of head hunters be universal. 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.

Internet Component Broker

The Internet Component Broker (ICB) acts as a mediator between two components adhering to different component models. The broker will utilize adapter technology, each adapter component providing translation capabilities for specific component architectures. Thus, a computational aspect of the adapter component will indicate the models for which it provides interoperability. It is expected that brokers will be pervasive in an Internet environment thus providing a seamless integration of disparate components. Adapter components will register with the ICB and while doing so they will indicate their specializations (which component models they can bridge efficiently). During a request from a seeker, the head hunter component will not only search for a provider, but it will also supply the necessary details of an ICB.

The adapter components achieve interoperability using the principles of *wrap* and *glue* technology [9]. A reliable, flexible and cost-effective development of wrap and glue is realized by the automatic generation of glue and wrappers based on component specifications. Wrapper software provides a common message-passing interface for components that frees developers from the error prone tasks of implementing interface and data conversion for individual components. The glue software schedules time-constrained actions and carries out the actual communication between components.

The functionality of the ICB is analogous to that of an object request broker (ORB). The ORB provides the capability to generate the glue and wrappers necessary for objects written in different programming languages to communicate transparently; 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. An ORB defines language mappings and object adapters. An ICB must provide component mappings and component model adapters. While the ICB conceptually provides the capabilities of existing bridges (COM-CORBA for example), the ICB will provide key features that are unique; it is designed to provide the auxiliary aspects of the Internet – collaboration between autonomous environments, mobility and security. In addition, the UMM includes quality of service and service guarantees. The ICB, in conjunction with head-hunters provide the infrastructure necessary for scalable, reliable, and secure collaborative business using the Internet.

3 A Case Study

In order to explain the UMM and the proposed approach, below a case study from the domain of distributed information filtering is presented. Although the case study uses a specific domain, the principles can be easily extended to other application domains that involve the software realization of a DCS.

3.1 Distributed Information Filtering

It is desired to develop a global information filtering system, in which, users will be interested in receiving selected information, based on their preferences, from scattered repositories. Usually, a filtering task involves contacting the scattered resources, performing an initial search to gather a subset of documents, representing, classifying and presenting based on the user profile. Many different methods are employed for the sub-tasks involved in filtering. Thus, it can be easily envisioned that different components, each employing a different algorithm to perform these sub-tasks, will be scattered across an interconnected system. Each component may belong to a different model, may quote different costs and offer different qualities of service.

Hence, a typical distributed information filtering system consists of the following types of components: a) Domain Component (DC), b) Wrapper Component (WC), c) Representer Component (RC), d) Classifier Component (CC), and e) User Interaction Component (UIC). In addition to these domain-specific components, headhunter components (HC) and the ICB are needed.

All these components, their aspects and characteristics need to be defined using UMM. For the sake of brevity, only the complete description of the domain component (DC) is shown below.

3.2 Domain Component

The domain component is responsible for maintaining a repository of URLs of associated information sources for particular type (e.g., text, structure, sequence) of information that needs filtering.

For example, the inherent attributes might consist of Author (name of the component developer), Version (current version of the component), Date Deployed, Execution Environment Needed and Component Model (e.g., Java-RMI 1.2.2), Validity (e.g., one month from the deployment), Atomic or Complex (indivisible or an amalgamation of other components, e.g. atomic), Registrations (with which headhunters this component is registered, e.g., H1 – www.cs.iupui.edu/h1 and H2 – www.cis.uab.edu/h2).

An informal description of the functional part of a component may contain:

1. Computational Task Description -- e.g., searching a selected set of databases over the Internet.
2. Algorithm Used and its Complexity -- Webcrawling and $O(n^2)$, respectively.
3. Alternative Algorithms -- Indexing.
4. Expected Resources (best, average and worst-cases) -- multi-processor, uni-processor (300MHz with an CPU utilization of 50%), and uni-processor (100MHz with CPU utilization of 99%), respectively.
5. Design Patterns Used (if any) -- Broker.
6. Known Usages -- for assembling an up-to-date listing containing addresses of known information repositories for a particular domain.
7. Aliases-- such a component is usually called a Pro-active Agent.
8. Multi-level contracts:
e.g., for a function like `List getURLs (Domain inputDomain, Compensation inputCost)`, the behavioral contract could specify the pre-condition to be (valid Domain Name and cost), post-condition to be: `if successful (activeClientThreads++ and cost+=inputCost)`
`else (raise DomainNotKnownException and InvalidCostException)`
and the invariant could be `(ListOfURLs > 1)`. Also, for the same function, the concurrency contract could specify (maximum number of active threads allowed = 50).

The cooperation attributes of the domain component may consist of 1) expected collaborators UIC, WC, HC, TC and RC, 2) pre-processing collaborators HC and TC, and 3) post-processing collaborators RC and UIC.

The auxiliary attributes of the domain component are 1) fault-tolerant attributes, e.g., check-pointing versions, 2) security attributes, e.g., simple encryption, and 3) mobility attributes, e.g., “not mobile.”

For the domain component, the QoS parameters may contain 1) number of available URL's, 2) ranking of URL's, and 3) average rate of URL collection.

A component developer may offer several possible levels of QoS, e.g., L1) novice (number of URL's < 50 and no ranking of URL's and average rate of URL collection ≥ 1 week and average latency ≥ 2 minutes), L2) intermediate (number of URL's < 500 and simple ranking of URL's and average rate of URL collection ≥ 3 days and average latency ≥ 1 minute), and L3) expert (number of URL's < 1500 and advanced ranking of URL's and average rate of URL collection ≥ 1 day and average latency ≥ 5 seconds).

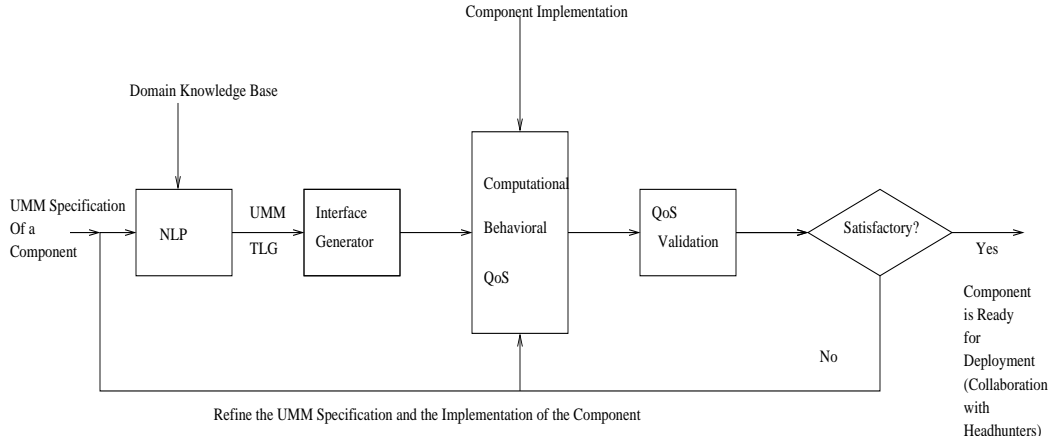


Figure 1: The Component Development and Deployment Process in UMM

The expected compensations for the above levels in terms of the number of URLs could be 1) L1 > 100 and < 200, 2) L2 > 200 and < 400, and 3) L3 > 400 and < 600.

4 Component and System Generation Using UMM Framework

The development of a software solution, using the UMM approach, for a DCS has two levels: a) component level – in this level, different components are created by developers, tested and verified from the point of view of QoS, and then deployed on the network, and b) system level – this level concentrates on assembling a collection of components, each with a specific functionality and QoS, and semi-automatically generates the software solution for the particular DCS under consideration. These two levels and associated processes are described below.

4.1 Component Development and Deployment Process

The component development and deployment process is depicted in Figure 1. As seen in the figure, this process starts with a UMM specification of a component (from a particular domain). This specification is in a natural-language format, as illustrated in the previous section. This informal specification is then refined into a formal specification. The refinement is based upon the theory of Two-Level Grammar (TLG) natural language specifications [3, 23], and is achieved by the use of conventional natural language processing techniques (e.g. see [7]) and a domain (such as information filtering) knowledge base. TLG specifications allow for the generation of the interface (possibly multi-level) for a component. This interface incorporates all the aspects of the component, as required by the UMM. The developer provides the necessary implementation for the computational, behavioral, and QoS methods. This process is followed by the QoS validation. If the results are satisfactory (as required by the QoS criteria) then the component is deployed on the network and eventually, it is discovered by one or more headhunters. If the QoS constraints are not met then the developer refines the UMM specification and/or the implementation and the cycle repeats.

4.2 Formal Specification of Components in UMM

Since the UMM specifications are informally indicated in a natural language like style, our approach is to translate this natural language specification into a more formal specification using TLG. TLG is a formal notation based upon natural language and the functional, logic, and object-oriented programming paradigms. The name “two-level” in Two-Level Grammar comes from the fact that TLG consists of two context-free grammars, one corresponding to a set of type declarations and the other a set of function definitions operating on those types. These type and function definitions are incorporated into a class which allows for new types to be created.

The type declarations of a TLG program define the domains of the functions and allow strong typing of identifiers used in the function definitions. On the other hand, function definitions may be given without precisely defined domains for a more flexible specification approach. This framework consists of a knowledge-base which establishes a context for the natural language text to be used in the specification under a particular domain model, in this case information filtering. This allows the TLG to be translated into internal representations such as predicate logic, the natural representation for TLG, event grammars, or multi-level Java interfaces taking the form of the UMM specification template. For the case

study, we may use a TLG class to describe the component structure and functionality as elaborated in the following subsections.

4.2.1 Component Structure Specification

Syntactically, TLG type declarations are similar to those in other languages. Types are capitalized whereas constants begin with lower case letters. The usual primitive types, such as `Integer`, `Float`, `Boolean`, and `String` are present as are list constructors based upon regular expression notation, e.g. `{X}*` and `{X}+` mean 0 or more and 1 or more occurrences of `X`, respectively.

The types of the domain component in our information filtering system are defined in the following way in TLG.

```
Component :: DomainComponent; WrapperComponent; RepresentationComponent; ClassificationComponent;
           UserInteractionComponent; HeadhunterComponent; ICB.
DomainComponent :: Name, InformalDescription, Attributes, Service.
Name :: dc.
Attributes :: ComputationalAttributes, CooperationAttributes, AuxiliaryAttributes.
ComputationalAttributes :: InherentAttributes, FunctionalAttributes.
InherentAttributes :: Author, Version, DateDeployed, ExecutionEnvironment,
                    ComponentModel, Validity, Structure, Registrations.
FunctionalAttributes :: TaskDescription, AlgorithmAndComplexity,
                    Alternatives, Resources, DesignPatterns, Usages, Aliases, FunctionsAndContracts.
AlgorithmAndComplexity :: webcrawling, n^2; ....
Alternatives :: {AlgorithmAndComplexity}*.
Resource :: Architecture, Speed, Load.
Architecture :: uni-processor; multi-processor.
Speed :: Integer.
Load :: Integer.
DesignPatterns :: broker; ....
Aliases :: pro-active agent; ....
FunctionAndContract :: Function, BehavioralContract, ConcurrencyContract.
Function :: ....
BehavioralContract :: Precondition, Invariant, Postcondition.
ConcurrencyContract :: single threaded; maximum number of active threads allowed = Integer; ....
CooperationAttributes :: ExpectedCollaborators, PreprocessingCollaborators, PostprocessingCollaborators.
ExpectedCollaborator :: uic; wc; hc; tc; rc.
PreprocessingCollaborator :: hc; tc.
PostprocessingCollaborator :: rc; uic.
AuxiliaryAttribute :: FaultTolerantAttribute; SecurityAttribute; MobilityAttribute.
FaultTolerantAttribute :: check-pointing versions; ....
SecurityAttribute :: simple encryption; ....
MobilityAttribute :: mobile; not mobile.
Service :: ExecutionRate, ParallelismConstraint, Priority, Latency, Capacity, Availability,
          OrderingConstraints, QualityOfResultsReturned, ResourcesAvailable, ....
ExecutionRate :: Float.
ParallelismConstraint :: synchronous; asynchronous.
Priority :: Integer.
Latency :: AverageRateOfURLCollection.
AverageRateOfURLCollection :: Float.
Capacity :: NumberOfAvailableURLs.
NumberOfAvailableURLs :: Integer.
Availability :: Float.
OrderingConstraint :: Boolean.
QualityOfResultsReturned :: {URL}+.
ResourcesAvailable :: HardwareResources, SoftwareResources.
HardwareResources :: ....
SoftwareResources :: ....
```

The remaining components (e.g., wrapper, representation, etc.) may be described in a similar manner. All domains not specified explicitly in the above example are assumed to be of type `String`, with the exception of `Function` which may take the form of an interface definition in a programming language such as Java. Using standard natural language processing techniques [7], the UMM specification may be automatically refined into this TLG specification, with user assistance as

needed to clarify ambiguities. The process is facilitated by the presence of a knowledge base which understands the domain of information filtering from the point of view of vocabulary which may be used in making the original UMM specification.

4.2.2 Component Functionality Specification

The second level of the TLG specification is for function declarations. These resemble logical rules in logic programming with variables coming from the domains established in the type declarations. For the Domain Component example, the levels of Quality of Service may be specified as follows.

```
number of urls : size of QualityOfResultsReturned.
average latency : ...
no ranking of urls : ...
simple ranking of urls : ...
advanced ranking of urls : ...
average latency : ...
qos level 1 is novice : number of urls < 50, no ranking of urls,
    AverageRateofURLCollection >= 1 week, average latency >= 2 minutes.
qos level 2 is intermediate : number of urls < 500, simple ranking of urls,
    AverageRateofURLCollection >= 3 days, average latency >= 1 minute.
qos level 3 is expert : number of urls < 1500, advanced ranking of urls,
    AverageRateofURLCollection >= 1 day, average latency >= 5 seconds.
```

Each rule defines how the particular entity is to be computed. As these rules are normally part of a class definition encapsulating a corresponding set of type declarations, each rule has access to the data specified in the type declarations. These natural language like rules may be further refined into a more formal specification, e.g. using event grammars.

4.3 QoS Guarantee of a Domain Component

For the case study, the event grammar to describe the system behavior is given below. The first part is the set of type definitions and the second part is the description of computations over event traces implementing different QoS metrics.

```
exec_syst :: (request_sent | response_received)*
response_received :: (URL_detected | failed)
```

These type definitions describe the types of events which may occur as the system executes. The computations over these events include verification that the number of URL's detected is less than 50 and also the latency (e.g., for all requests for URL's, every response received occurs within 10 units of time). `id` is an event attribute which associates a unique identifier between query attributes and corresponding responses. Both of these metrics yield Boolean values.

```
CARD [URL_detected from exec_syst] < 50
forall x : request_sent from exec_syst
  Exists y : response_received from exec_syst
    id (x) = id (y) & begin_time (y) - end_time (x) < 10
```

4.4 Automated System Generation and Evaluation based on QoS

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, then the task is to assemble them. Figure 2 shows a process to accomplish this. 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. For example, such a query might be a request to assemble an information filtering system. The natural language processor (NLP) processes the query. It does this aided by the domain knowledge (such as key concepts in the filtering domain) and a knowledge-base containing the UMM description (in the form of a TLG) of the components for that domain. The result is a formal UMM specification that will be used by headhunters for component searches and as an input to the system assembly step. This formal UMM specification will be a basis for generating a set of test cases to determine whether or not an assembly satisfies the desired QoS. The framework, with the help of the infrastructure described in Section 2.2.3, collects a set of potential components for that domain, each of which meets the QoS requirement specified by the developer. From these, the developer, or a program acting as a proxy of the developer, selects some components. These components along with the component broker and appropriate adapters (if needed) form a software implementation of the distributed system. Next this implementation is tested using event traces and 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

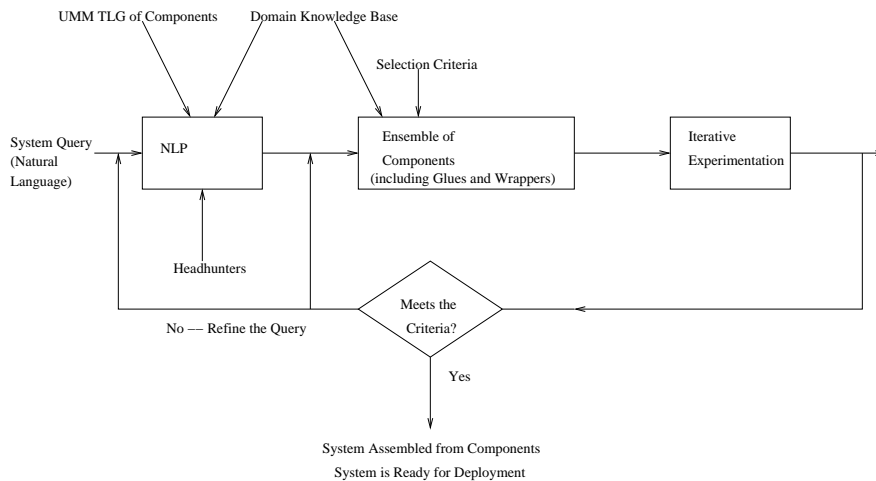


Figure 2: The Iterative System Integration Process in UMM

components or it may attempt to refine the query by adding more information about the desired solution from the problem domain. Once a satisfactory implementation is found, it is ready for deployment.

5 Conclusion

This paper has presented a framework that allows an interoperation of heterogeneous and distributed software components. The software solutions for future DCS will require either automatic or semi-automatic integration of software components, while abiding with the QoS constraints advertised by each component and the collection of components. The result of using UMM and the associated tools is a semi-automatic construction of a distributed system. Glue and wrapper technology allows a seamless integration of heterogeneous components and the formal specification of all aspects of each component will eliminate ambiguity while detecting and using these components. The UMM does not consider network failures or other considerations related to the hardware infrastructure, however, these could be integrated into the QoS level of components. The UMM approach to validating QoS is to use event grammar to calculate QoS metrics over run-time behavior. The QoS metrics are then used as a criteria for an iterative process of assembling the resulting system as shown in Section 4.4. UMM also provides an opportunity to bridge gaps that currently exist in the standards arena. Although, the paper has only presented a case study from the domain of distributed information filtering, the principles of UMM may be applied to other distributed application domains. Future work includes refinement of the UMM feature thesaurus and methods for translating UMM specifications into Two-Level Grammar, refining the head-hunter mechanism, developing Quality of Service metrics for components and systems, and development of generation mechanisms for domain-specific component reuse.

References

- [1] Auguston, M. A Language for Debugging Automation. In *Proceedings of 6th International Conference on Software Engineering and Knowledge Engineering*, pages 108–115, 1994.
- [2] Beugnard, A., Jezequel, J., Plouzeau, N. and Watkins, D. Making Components Contract Aware. *IEEE Computer*, 32(7):38–45, July 1999.
- [3] Barrett R. Bryant. 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.
- [4] California Institute of Technology. *Caltech Infospheres On-line Documentation*, URL:- <http://www.infospheres.caltech.edu/>, 1998.
- [5] Fox, G. The Document Object Model Universal Access Other Objects CORBA XML Jini JavaScript etc. <http://www.npac.syr.edu/users/gcf/msrcojectsapril99>, 1999.
- [6] Israel, B. and Kaiser, G. Coordinating Distributed Components Over the Internet. *IEEE Internet Computing*, pages 83–86, 2(2), 1998.
- [7] Jurafsky, D. and Martin, J. H. *Speech and Language Processing*. Prentice Hall, 2000.

- [8] Kotz, D., Gray, R., Nog, S., Rus, D., Chawla, S. and Cybenko, G. Agent TCL: Targetting the Needs of Mobile Computers. *IEEE Internet Computing*, pages 58–67, 1(4), 1997.
- [9] 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.
- [10] Manola, F. Technologies for a Web Object Model. *IEEE Internet Computing*, 3(1):38–47, January-February 1999.
- [11] Microsoft Corporation. *DCOM Specifications*, URL:- <http://www.microsoft.com/oledev/olecom>, 1998.
- [12] Object Management Group. XML Metadata Interchange. Technical report, Object Management Group Document No. ad/98-10-05, October 1998.
- [13] Object Management Group. CORBA Components. Technical report, Object Management Group TC Document orbos/99-02-05, March 1999.
- [14] Object Management Group. Meta Object Facility (MOF) Specification, Version 1.3. Technical report, Object Management Group, March 2000.
- [15] Object Management Group. Model Driven Architecture: A Technical Perspective. Technical report, Object Management Group Document No. ab/2001-02-01, February 2001.
- [16] Orfali R, and Harkey, D. *Client/Server Programming with JAVA and CORBA*. John Wiley & Sons, Inc., 1997.
- [17] 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.
- [18] Rogerson, D. *Inside COM*. Microsoft Press, 1996.
- [19] Rumbaugh, J., Jacobson, I. and Booch, G. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.
- [20] Siegel, J. *CORBA Fundamentals and Programming*. John Wiley & Sons, Inc., 1996.
- [21] The Globus Project. *Globus Website*, URL:- <http://www.globus.org/>, 2000.
- [22] University of Virginia. *Legion Project*, URL:- <http://www.cs.virginia.edu/legion>, 1999.
- [23] van Wijngaarden, A. Orthogonal Design and Description of a Formal Language. Technical report, Mathematisch Centrum, Amsterdam, 1965.