A Method for Design and Performance Modeling of Client/Server Systems

Daniel A. Menascé, *Member*, *IEEE Computer Society*, and
Hassan Gomaa, *Member*, *IEEE Computer Society*

Abstract—Designing complex distributed client/server applications that meet performance requirements may prove extremely difficult in practice if software developers are not willing or do not have the time to help software performance analysts. This paper advocates the need to integrate both design and performance modeling activities so that one can help the other. We present a method developed and used by the authors in the design of a fairly large and complex client/server application. The method is based on a software performance engineering language developed by one of the authors. Use cases were developed and mapped to a performance modeling specification using the language. A compiler for the language generates an analytic performance model for the system. Service demand parameters at servers, storage boxes, and networks are derived by the compiler from the system specification. A detailed model of DBMS query optimizers allows the compiler to estimate the number of I/Os and CPU time for SQL statements. The paper concludes with some results of the application that prompted the development of the method and language.

Index Terms—Software performance engineering, performance models, client/server systems, queuing networks, database query optimization, UML, CLISSPE.

1 Introduction

An increasing number of organizations are moving mission-critical applications from mainframe environments to client/server (C/S) systems. Designing distributed C/S applications that meet performance requirements is not a trivial task for complex and distributed C/S systems. There are often a large number of software and hardware architectural choices to be made when designing a C/S system. It is usually not clear what the impact on performance is of the various choices. Examples of these choices include the distribution of work between client and server, use of three-tiered C/S architectures, distribution of functions among servers, distribution of database tables among servers, type of client and servers, and network connectivity. Waiting until the application is ready to go into production is not a viable option, since poor performance may require major code redesign and rewrite. This is usually very expensive in terms of development cost and cost incurred by a delayed deployment of the new application.

To ensure that the new application will meet the performance requirements, software performance engineering (SPE) [11], [22], [27], [41] techniques have to be employed during the software design and development process. These techniques estimate the demands of the new application and use performance models to predict the performance of the new system.

- *D.A. Menascé is with the Department of Computer Science, George Mason University, Fairfax, VA 22030-4444. E-mail: menasce@cs.gmu.edu.*
- *H. Gomaa is with the Information and Software Engineering Department, George Mason University, Fairfax, VA 22030-4444. E-mail: hgomaa@gmu.edu.*

Manuscript received 6 Apr. 1999; revised 7 Oct. 1999; accepted 15 Mar. 2000. Recommended for acceptance by A. Cheng, P. Clements, and M. Woodside. For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 111941.

This paper advocates the need to integrate both design and performance modeling activities, so that one can help the other. We describe an iterative approach for designing a system and modeling its performance before it is implemented. The goal is to analyze the design from a performance perspective, to compare alternative designs, and to compare the design executing on different system configurations. To do this, it is necessary to model the design at the level of granularity of message communication between client and server and model the application functionality at the client and server side to capture the application logic and pattern of access to the system resources. In a relational database intensive C/S application, it is necessary to explicitly model the relations used and the access patterns to these relations by the application.

The design method described in this paper is an object-oriented approach based on use cases, a structural view using object models, and a dynamic view using object collaboration diagrams and is based on the concepts of Jacobson and Rumbaugh. When the project was carried out, we used the earlier notations given in [15], [36]. For the purposes of this paper, we have used the newer UML notation [1], [14], [35], [10].

The performance modeling aspect of the method is based on a language, called CLISSPE (Client/Server Software Performance Evaluation) [22], that can be used by software developers to specify use cases and by performance analysts for software performance prediction. The need for this method was prompted by the involvement of the authors in the redesign and capacity planning of a very large mission-critical application.

The CLISSPE language allows designers of C/S systems to describe different kinds of objects such as servers, clients, databases, relational database tables, transactions, and networks, as well as the relationship between them. Examples of

relationships include mappings of servers and clients to networks and mappings of database tables to servers. The language also allows the designer to specify the actions executed by each transaction. A CLISSPE specification compiles into an analytic queuing network model for the C/S system allowing for the capacity planning of the application under development. Service demand parameters at servers, storage boxes, and networks are derived by the compiler from the system specification. A detailed model of DBMS query optimizers allows the compiler to estimate the number of I/Os and CPU time for SQL statements.

The rest of this paper is organized as follows: Section 2 presents a brief overview of SPE and the elements of the CLISSPE language. Section 3 discusses the application that motivated the development of the method presented here and the design of the CLISSPE language. Section 4 provides an overview of the method. Sections 5 through 12 provide details of each step of the method. Section 13 provides the analytic models used by the CLISSPE compiler to determine service demands due to database accesses. Section 14 discusses the parameter gathering activity for the project that motivated this study. A few results of this study are discussed in Section 15. Section 16 presents a discussion of our approach and elaborates on future work. Finally, some concluding remarks are given in Section 17.

2 SPE AND CLISSPE

SPE requires that performance models be built and solved to predict the performance of the new software system. In the method described in this paper, we use queuing network models to predict the performance of software systems under development. These models require two types of parameters: workload intensity (e.g., transaction arrival rates) and service demands at the various resources including server CPUs, I/O subsystems, LAN segments, and WANs. While workload intensity parameters can be usually obtained from the performance requirements of the software under development, the same is not true for service demands. Obtaining estimates of service demands requires a thorough understanding of the applications business rules as well as the design of the databases used to support the application. Obtaining this knowledge may prove to be extremely difficult in practice if software developers are not willing or do not have the time to help software performance analysts. This may be one of the biggest challenges faced by software performance analysts who need the collaboration of software developers to obtain input parameters for their models.

In our case, the task of estimating service demands was accomplished with the CLISSPE language [22]. CLISSPE has three sections: a declaration section, a mapping section, and a transaction specification section. The declaration section is used to declare the following objects: clients and client types, servers and server types, disks and disk types, database management systems, database tables, networks and network types, transactions, remote procedure calls (RPCs), and numeric constants.

The mapping section is used to allocate clients to networks, allocate servers to networks, assign transactions to clients, specify network paths (from clients to servers going through several networks), and assign database tables to servers. Finally, the transaction specification section is used to specify the logic of each of the major transactions. This specification is oriented towards software performance engineering. Therefore, loop specifications indicate the average number of times a loop is executed, branch statements indicate the probability that a certain path is followed, and case statements indicate the probability each option is executed. A complete description of the language can be found in [23].

Several commercial tools such as SPE.ED, QASE, SES/workbench, and others can be used for software performance engineering. Some of these tools are based on simulation while some use simulation and analytic models. Most provide a graphical interface for specifying hardware and software systems. We decided to develop our own set of tools for the study at hand since 1) we would have more control over the underlying models used; for example, the CLISSPE system models DBMS query optimizers at a considerable level of detail, 2) it would not require us to go through the learning curve associated with the adoption of new tools, and 3) we wanted the software designers to use the CLISSPE language to specify their use cases. CLISSPE can be used by both software system designers and performance engineers. One of the major deterrents for the widespread use of SPE is that SPE is viewed by many as an activity separate from software design and development and, therefore, should be carried out by people with different skills. With our language-oriented approach, we strove to bridge the gap between these two camps. In fact, in the project that prompted this study, all CLISSPE programs were written by a system designer who is not an expert in performance.

Significant activity in software performance engineering has taken place recently. A recent workshop on software and performance brought together the performance and software communities and resulted in a very lively interchange of ideas [40].

3 MOTIVATING APPLICATION

The application that prompted the development of the method presented in this paper is a Recruitment and Training System (RTS) of a major US Government agency. The current system is being downsized from a mainframe-based system to a C/S environment. Applicants go to recruitment centers spread all over the country. There, personnel specialists interview the applicants and try to match the applicant skills with the agency's desired skills. Accepted applicants are recruited and are assigned to one or more training classes where they will acquire the skills needed for the job.

The current application and databases reside on an aging mainframe that is expensive to maintain. The current application has a line-oriented user interface and is difficult to maintain because many programs are more than 20 years old and the application is written in many different programming languages. Also, due to its centralized nature, the current system does not scale well with the number of users. The new environment, shown in Fig. 1, is composed

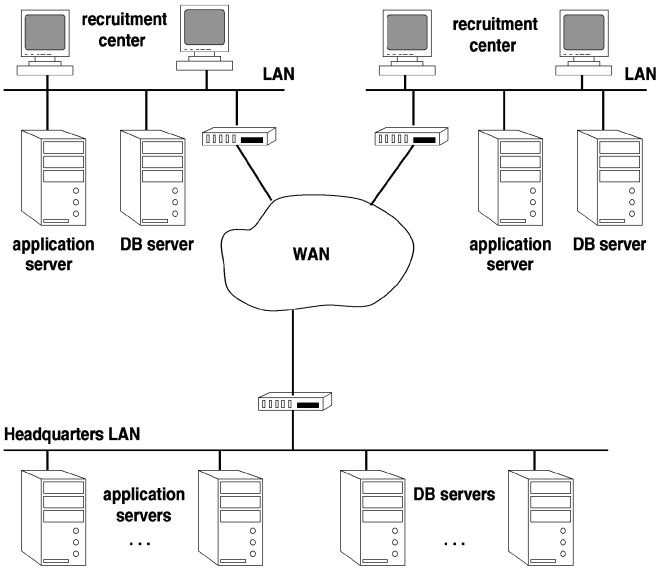

Fig. 1. Client/Server configuration for RTS.

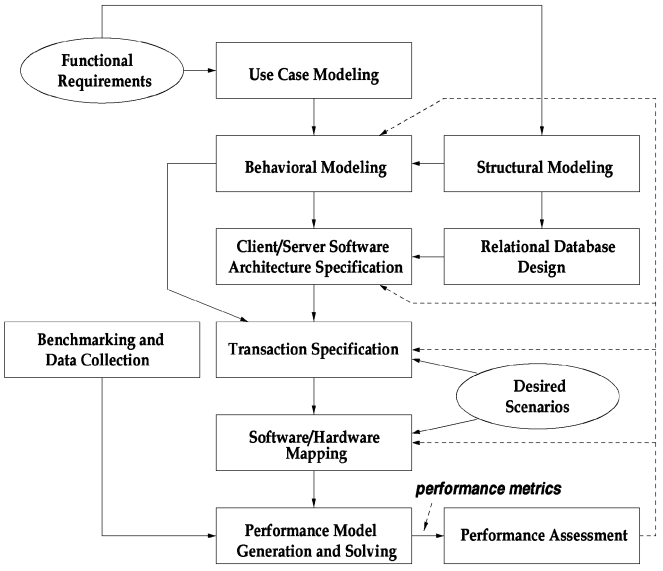

Fig. 2. Integrated software architectural design and performance analysis method.

of several recruitment centers where several client workstations are interconnected through a 10-Mbps Ethernet LAN. Each recruitment center may or may not have a local application server and a local database server. Recruitment centers are connected through a Wide Area Network (WAN) to the headquarters LAN where one or more application and database servers are located.

The current application is supported mainly by VSAM files. The new C/S version is based on a database with close to 200 tables supported by Oracle. Tuxedo is being used as a Transaction Processing Monitor (TPM).

4 Overview and Approach

4.1 Overview of UML

Object-oriented concepts are considered important in software reuse and evolution because they address fundamental issues of adaptation and evolution. Object-oriented methods are based on the concepts of information hiding, classes, and inheritance. Information hiding [31], [30] can lead to more self-contained and, hence, modifiable and maintainable systems. Inheritance [35] provides an approach for adapting a class in a systematic way. With the proliferation of notations and methods for the object-oriented analysis and design of software systems, the Unified Modeling Language (UML) is an attempt to provide a standardized notation for describing object-oriented models. However, for the UML notation to be effectively used, it needs to be used in conjunction with an object-oriented analysis and design method.

The Object Oriented Analysis and Modeling method used in this project employed a combination of use cases [15], object modeling [36], statecharts [12], [36], and event sequence diagrams used by several methods [10], [14], [36]. The notation used is based on the Unified Modeling Language (UML) [1], [10], [35]. In use case modeling, the functional requirements of the system are defined in terms of use cases and actors. Structural modeling provides a static view of the information aspects of the system. Classes are defined in terms of their attributes, as well as their relationship with other classes. Behavioral modeling provides a dynamic view of the system. The use cases are refined to show the interaction among the objects participating in each use case. Object collaboration diagrams and sequence diagrams are developed to show how objects collaborate with each other to execute the use case. The state dependent aspects of the system are defined using statecharts. In particular, each state dependent object is defined in terms of its constituent statechart.

4.2 Approach

This section provides an overview of the iterative, integrated, object-oriented method for the design and performance analysis of Client/Server systems (see Fig. 2). The steps of the method are briefly described below, with a more detailed description given in the ensuing sections:

1. **Define Use Case Model.** The functional requirements of the system are specified in terms of use cases. A use case [15] is a scenario describing the interaction between the user (referred to as an actor) and the system. The use cases are depicted on a use case diagram.
2. **Define Structural Model.** The structural model [36] provides an information modeling perspective on the system. It defines the classes in the system, the attributes of each class, the operations of each class, and how the classes are related to each other. The classes are depicted on a class diagram.
3. **Define Behavioral Model.** The use cases are expanded to describe what objects participate in each use case and the sequence of object interactions. The objects participating in each use case are depicted on an object collaboration diagram.
4. **Map Structural model to relational database.** The classes in the structural model are mapped to relations in a relational database.

5. **Develop the Client/Server Software Architecture.** The C/S software architecture is designed. The goal is to have a configurable message based design that allows objects to be mapped to client or server nodes.
6. **Develop the Transaction Specification.** The transactions are specified in the CLISSPE language.
7. **Define Software/Hardware Mapping.** The C/S software architecture is mapped to a specific system configuration.
8. **Performance Modeling and Assessment.** The performance model is based on queuing network models [27]. A discussion on analytic models of software systems is outside the scope of this paper. Several methods such as Layered Queuing Models [34], the Rendez-vouz method [43], and approximations to simultaneous resource possession [20], [25] for Mean Value Analysis [33] can be used. Other models of C/S systems can be found in [3], [32]. The outputs of the performance model include response times and throughputs for each type of request submitted to the system. An analysis of the results of the performance model reveals the possible bottlenecks. If the C/S configuration does not meet the performance objectives, architectural changes at the hardware and/or software level have to take place. These changes, guided by the outputs of the performance model, may be applied to the C/S software software architecture, the transaction specification, the software/hardware mapping, or possibly even the refined use cases. Successive iterations ensure that the final design meets the performance objectives.

5 Step 1: Use Case Modeling

The functional requirements of the system are specified in terms of use cases. A use case is a scenario describing the interaction between the user (referred to as an actor) and the system. In the requirements phase, the use case considers the system as a black box and describes the interactions between the user and the system in a narrative form consisting of user inputs and system responses. Use cases may be structured using the Uses and Extends concepts to maximize extensibility and reuse. This step follows Jacobson's use-case driven approach [14], [15].

Abstract use cases reflect functionality that is common to more than one use case. Analysis of several use cases can reveal common parts among these use cases. By separating out this common functionality into an abstract use case, the abstract use case and the objects that participate in it, can be reused by several concrete (executable) use cases.

Consider an example of use cases from the training system (see Fig. 3).

The actor is the personnel specialist who is the main user of the training system. There are two types of people who need to be trained, a new applicant or an existing employee. For each type of trainee, a concrete use case is developed, Check New Applicant and Check Existing Employee. Each concrete use case uses several abstract use cases, allowing some abstract use cases to be used by both types of user.

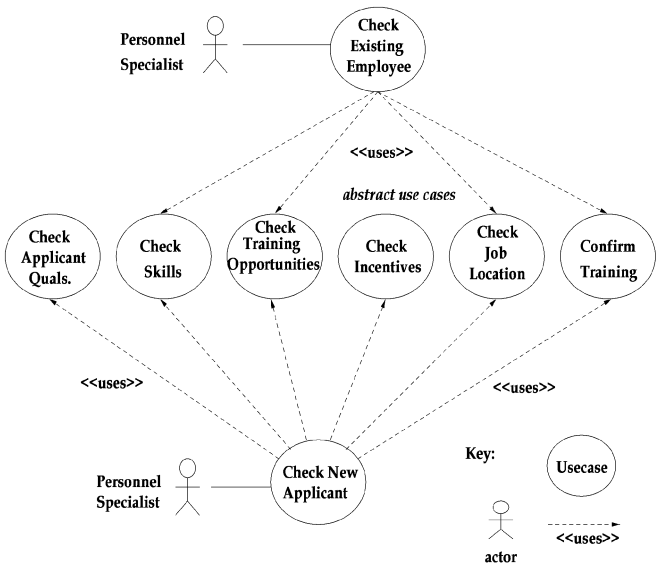

Fig. 3. Training system use case diagram.

Consider the new applicant. The sequence of steps to process a new applicant is:

- **Check New Applicant Qualifications.** Check that the applicant has the minimum qualifications for acceptance into a training course.
- **Check Skills (new applicant and existing employee).** Checks the person's skills and, hence, determine what new skills he/she is ready to be trained for.
- **Check Training Opportunities (new applicant and existing employee).** Check the availability of training courses for the skills the applicant is qualified to pursue.
- **Check Incentives.** Check incentives available for qualified new applicants.
- **Check Job Location (new applicant and existing employee).** When training is complete, trainee has choice of locations for available jobs.
- **Confirm Training (new applicant and existing employee).** Person selects a training course and receives a confirmation.

Of the six abstract use cases, two are specific to new applicants, while four are also used to process existing employees. The complete specification of the Check Skills use case is given below:

Use Case: Check Skills

Summary: This use case checks the applicant's (new applicant or existing employee) skills and, hence, determines what new skills he/she is ready to be trained for.

Actors: Personnel Specialist.

Precondition: Applicant's record exists.

Description: Personnel Specialist enters the applicant's social security number. The system determines the skills that the applicant possesses. It then determines what new skills the applicant is allowed to train for by

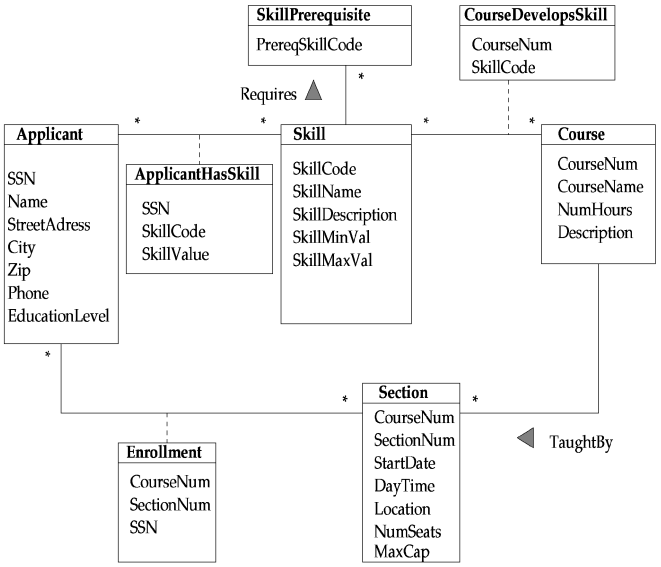

Fig. 4. Structural model for training system.

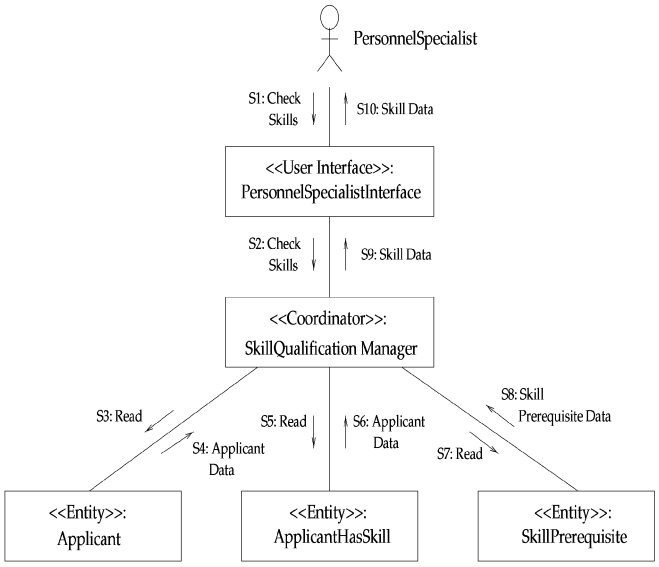

Fig. 5. Object collaboration diagram for check skills use case.

checking whether the applicant's skills match the prerequisite skills of the new skill. The system displays all skills the applicant is qualified to train for.

Alternatives: Applicant does not qualify for training for any new skills.

Postcondition: List of skills displayed to Personnel Specialist.

6 Step 2: Structural Modeling

The Structural Model (also known as the Object Model) addresses the static structural aspects of a problem by modeling classes in the real world [36]. A Structural Model describes the static structure of the system being modeled, which is considered less likely to change then the functions of the system. The origins of the Structural Model are in information modeling, in particular entity-relationship modeling, as used in logical database design.

A Structural Model defines the classes in the system, the attributes of the classes, the relationships between classes, and the operations of each class. Three types of relationships are possible: associations, composition/aggregation, and generalization/specialization.

In the analysis phase, a conceptual structural model is developed, which concentrates on the entity classes that will eventually be mapped to a database. Entity classes are persistent long-lasting classes that store information. An entity object is typically accessed by many use cases. The information maintained by an entity object persists over access by several use cases.

As an example, consider the entity classes of the training system (see Fig. 4), which shows entity classes, attributes, associations, and association classes. The Structural Model shows that an Applicant has one or more Skills.

An association class is a class that models an association between classes. The association class ApplicantHasSkill defines the specific Applicant Skill association. A Skill has 1 or more PrerequisiteSkills. Each Skill uses one or more Courses to train for it. A Course is taught by one or more Sections. The Applicant Section association is many-many and, so, an association class Enrollment defines the specific Section a given Applicant is enrolled in.

7 Step 3: Behavioral Modeling

The behavioral model, also referred to as the dynamic model, shows the dynamic aspects of the system. During behavioral modeling, the following steps need to be performed. For each use case developed during use case modeling, the objects that participate in the use case are determined as well as the sequence of message interactions between the objects. State dependent objects are defined using statecharts, although this feature was not used for this study. The sequence of object interactions is depicted on an *object collaboration diagram*. This form of object-oriented behavioral modeling is different from user behavioral modeling used to capture the frequency and characteristics of user interactions with the system. Examples of the latter type of models can be found in [13], [24].

Objects are structured using object structuring criteria and depicted on the object collaboration diagrams using the UML stereotype notation. In addition to entity objects, objects may be categorized as interface objects or control objects. It should be pointed out that the objects modeled are application-level objects. Thus, a user interface object is modeled as one object even though it may be an aggregate object that contains several lower-level GUI objects.

For the training reservation system, an object collaboration diagram (OCD) is developed for each use case. Two object collaboration diagrams are shown for the Check Skills and Check Training Opportunities use cases. In the OCD for the Check Skills use case (see Fig. 5), there is a user interface object, which is the interface to the actor, the Personnel Specialist. The entity objects are instances of the classes depicted on the class diagram. There is a coordinator object at the server, the Skills Qualification Manager, which

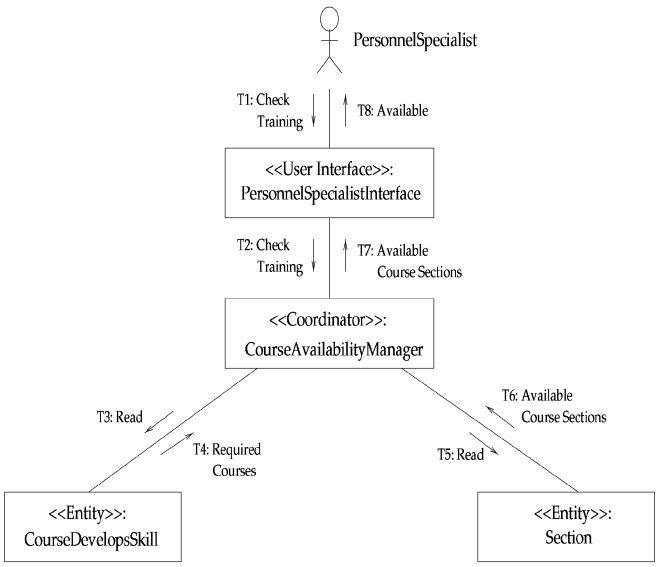

Fig. 6. Object Collaboration Diagram for Check Training Opportunities Use Case.

is the overall decision maker that provides the sequencing for the other objects participating in the use case.

Each object collaboration diagram has an associated message sequence description, which describes the sequence of messages. The message sequence description is a refinement of the black box description given in Section 5. For the Check Skills object collaboration diagram (see Fig. 5), the message sequence description is given next:

S1: Personnel Specialist enters the applicant's social security number and requests to check the applicant's skills.

S2: The Skills Qualification Manager receives the client request and uses the social security number to retrieve the applicant's data.

S3, S4: The Entity Applicant receives a Query request from the Skills Qualification Manager and responds by returning the applicant data.

S5, S6: The Skills Qualification Manager queries the Applicant Has Skill association object to determine the applicant's skills.

S7, S8: The Skills Qualification Manager queries the Skill Prerequisite object to determine what new skills the applicant is allowed to train for, by checking whether the applicant's skills match the prerequisite skills of the new skill.

S9: The Skills Qualification Manager returns a list of skills that the applicant is qualified to train for.

S10: The Personnel Specialist Interface object displays all skills the applicant is qualified to train for.

An object collaboration diagram for a subsequent use case, Check Training Opportunities, is given in Fig 6.

8 Step 4: Relational DB Design

As an Oracle relational database is to be used for the RTS system, the classes in the structural model are mapped to relations in a relational database. In particular, each class is mapped to a relation. Associations are mapped to either foreign keys or association relations. Additional relations may be needed if the classes are not normalized. For more information on mapping classes to relations, refer to [36].

For the classes depicted in Fig. 4, the relations are as follows, where an underlined attribute is a primary key:

Applicant (SSN, Name, StreetAddress, City, Zip, Phone, EducationLevel)
Skill (SkillCode, SkillName, SkillDescription, SkillMinVal, SkillMaxVal)
SkillPrerequisite (SkillCode, PrereqSkillCode)
Course (CourseNum, CourseName, NumHours, Description)
Section (CourseNum, SectionNum, StartDate, DayTime, Location, NumSeats, MaxCap)
ApplicantHasSkill (SSN, SkillCode, SkillValue)
CourseDevelopsSkill (CourseNum, SkillCode)
Enrollment (CourseNum, SectionNum, SSN)

The database relations, the existence of indices and their types, as well as information about column cardinality and selectivity is given in the CLISSPE specification of the C/S system. The specification in CLISSPE of tables Applicant and Enrollment is given below.

```
! declaration of database table Applicant
table Applicant num_rows= 1000000 row_size=
                120 dbms=Oracle
      columns= (SSN, Name, StreetAddress,
                City/200, Zip/99999, Phone,
                EducationLevel/10)
      index= (key= (SSN) key_size= 9 btree
              clustered)
      index= (key= (city) key_size= 20 btree)
      index= (key= (zip) key_size= 5 btree);
! declaration of database table Enrollment
table Enrollment num_rows= 400000 row_size=
                20 dbms= Oracle
      columns= (CourseNum/1000,
                SectionNum/10,SSN)
      index= (key= (coursenum, SectionNum, SSN)
              key_size= 18 btree clustered);
```

A table declaration specifies the average number of rows, the row size in bytes, the DBMS used to access the table, the DB table columns, and the indexes if any. Only columns referenced in a CLISSPE program need to be declared in a table declaration statement. The `columns=` parameter is used to provide the list of columns of the table. After each column name, an optional number following a / provides the column *cardinality* defined as the number of different values for the column present in the table. If the column cardinality is not provided, it is assumed to be equal to the number of rows in the table. The *selectivity factor* of a column is computed as the inverse of the cardinality, assuming that all values of each column are uniformly distributed. For example, in table Applicant, there are 200 different values in the column City in the table's 1,000,000 rows. Zero or more indexes may be declared for each table. An index key may be composed of the

concatenation of one or more columns. The key size in bytes is given by the parameter `key_size=`. The type of index, *hash* or *btree*, has to be specified. The optional keyword *clustered* indicates whether a b-tree index is clustered or not. At most one clustered index may be declared per table. See [29] for a good discussion on basic database concepts and query optimization.

9 Step 5: C/S Software Architecture Mapping Specification

The client/server software architecture is designed. The goal is to have a configurable message-based design that allows objects to be mapped to client or server nodes. This approach provides the flexibility of mapping the software architecture to different C/S configurations including two-tier or three-tier C/S configurations. In the two-tier configuration, user interface and application functionality is provided on the client nodes and the server node is a database server. In the three-tier configuration, the client nodes have the user interface functionality, while the application functionality is supported on an application server. The third tier is a database server, which can be configured to be on the same node as the application server or on a separate node. Furthermore, the database server may be configured to reside on one node or be distributed among multiple nodes. This flexible software architecture provides a framework for experimenting with different client/server system configurations, which can then be analyzed from a performance perspective.

For the objects depicted in Figs. 5 and 6, in a two-tier client/server configuration, all the objects would reside at the client and the server is the database server. In a three-tier client/server configuration, the user interface objects reside at the client node while the coordinator and entity objects reside at the application server node. The third tier is the database server node. The actual mapping from the client/server software architecture to a specific system configuration is done in the Software/Hardware Mapping step.

10 Step 6: Transaction Specification

The transaction business logic is specified in the CLISSPE language. The logic for the transaction is derived from the sequence of interactions depicted on the object collaboration diagram for a given use case and described in the message sequence description. The transaction references the relations defined in Step 4 of the method. The transaction has a client part and a server part to it as given by the Client/Server Software Architecture Mapping.

The client part of the transaction corresponds to the user interface object shown on the object collaboration diagram. The server part of the transaction corresponds to the other objects, namely the control and entity objects.

An example of the transaction specification for the Check Skills transaction is given next, which determines the skills that the applicant is qualified to train for. First is the specification for the client, which issues a remote procedure call to the application server.

```
transaction CheckSkills running_on client
  ! Actor enters applicant SSN
  ! check applicant skills
  rpc check_skills to_server ApplicServer;
  ! Display skills applicant is qualified to
   train for
end_transaction;
```

The application server follows the sequence of steps described in the object collaboration diagram for Check Skills (see Fig. 5). First the applicant relation is accessed and then a join is performed on the ApplicantHasSkill and SkillPrequisites relations. The value of the constant #ProbabilityApplicantWithMinSkills gives the probability that this branch is taken.

```
transaction CheckSkills running_on
                   server ApplicServer
 ! retrieve applicant's data
 select from Applicant where SSN;
 ! if applicant exists check applicant skills
 if #ProbabilityApplicantWithMinSkills
 then ! find all skills applicant qualifies
      ! to train for
      select from ApplicantHasSkill
                where SSN
             from SkillPrequisites
                where PrereqSkillCode
             joined_by
               ApplicantHasSkill.SkillCode =
                 SkillPrequisites.
                  PrereqSkillCode;
 end_if;
 ! Return skills applicant is qualified to train
 for
end_transaction; ! CheckSkills
```

A second transaction is the CheckTraining, which also has client and server modules. This transaction is executed after the Check Skills.

```
transaction CheckTraining running_on client
 ! check training opportunities
 ! Actor enters skills applicant qualified
 ! to train for
 rpc check_training to_server ApplicServer;
end_transaction;
```

The server loops for each skill and for the average number of courses per skill, then determines the courses required for this skill, loops on the average number of sections that must be checked for the course before a course is found. This corresponds to the OCD in Fig. 6.

```
transaction CheckTraining running_on
                     server ApplicServer
 ! Loop for training opportunities
 loop #average_skill_count
  !Determine courses required for this skill
  loop #average_num_courses
```

```
    select from CourseDevelopsSkill
           where SkillCode;
    ! Check sections for this course
    select from Section where CourseNum;
    loop #avg_sections_count
     ! Check if section is available
    end_loop;
    if #ProbSectionAvailable
    then !add to list of available opportunities
    end_if;
   end_loop;
  end_loop;
end_transaction; ! CheckTraining;
```

The transactions shown above contain examples of constants (e.g., #average_skill_count). Constants in CLISSPE start with a "#" character and are defined in the declaration section.

11 Step 7: Software Hardware Mapping

Given the desired client/server system configuration scenarios, the client/server software architecture is mapped to a specific system configuration that assigns software components to physical elements such as processors and network segments. The components of the system architecture are assigned performance characteristics (e.g., network segment speeds, router latencies, I/O subsystem bandwidth, processor speeds).

A few examples of how this type of mapping is specified in CLISSPE are given in what follows: In the declaration section of a CLISSPE C/S system specification, elements such as servers, client groups, database management systems, database tables, networks, and transaction types are declared. In the mapping section of the language, these elements are mapped to one another. For example, servers and client groups are mapped to networks, database tables are mapped to servers and disks within the servers, and database tables are assigned to database management systems.

Consider an example of a database server declaration and its mapping to a network. The example shows that server DBServer is declared as being of type IBM-RS-6000-M43P133 (this type has to be previously declared). The declaration of the DB Server indicates that Oracle is the DBMS running on it with a buffer size of 8,192 KBytes and configured to run on two CPUs and two disks. A network type HQType is defined as being a 100-Mbps Fast Ethernet. The Headquarters LAN, HQLan, is declared as being of this type.

```
! this goes in the declaration section
server DBServer type= IBM-RS-6000-M43P133
     dbms= Oracle DB_BuffSize= 8192 num_CPUs= 2
     disk dsk01 type= ServerDisk
     disk dsk02 type= ServerDisk;

network_type HQType bandwidth= 100
                    type= Fast_Ethernet;
network HQLan   type= HQType;
```

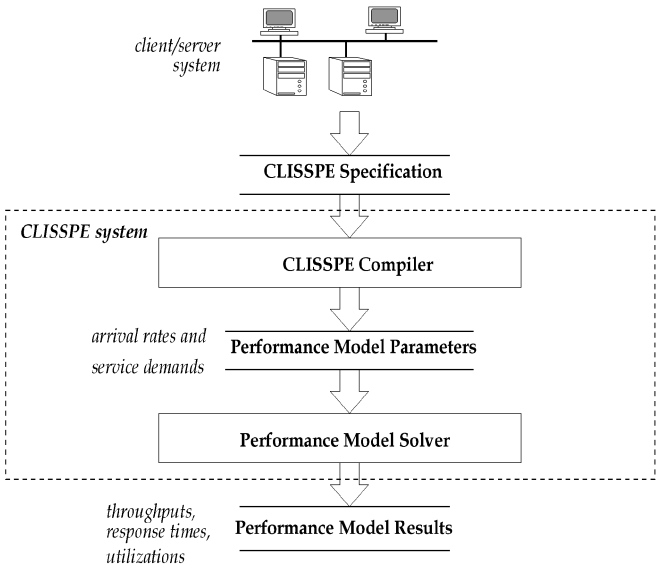

Fig 7. The CLISSPE system.

In the mapping section, the server DBserver is mapped to the network HQLan and table Applicant to the server DBServer. Sixty percent of the table Applicant is declared as being stored on disk one at that server and the remaining 40 percent at disk two.

```
! this goes in the mapping section
server DBServer is_in network HQLan;
table Applicant is_in server DBServer
      (dsk01: 0.6, dsk02: 0.4);
```

12 Step 8-a: Performance Modeling

The performance modeling step of the method is accomplished with the help of the CLISSPE system (see Fig. 7), which is composed of a CLISSPE compiler and a Performance Model Solver.

The CLISSPE compiler generates a multiclass open queuing network (QN) model [27] that corresponds to the specification given in CLISSPE. The QN model can be formally described as $\mathcal{Q} = (\mathcal{R}, \mathcal{W}, \vec{\lambda}, D)$ where $\mathcal{R}$ is the set of resources in the QN model that correspond to the various elements of the C/S system (e.g., processors, LAN segments, WANs, storage boxes), $\mathcal{W}$ is the set of workload classes, $\vec{\lambda} = (\lambda_1, \cdots, \lambda_{|\mathcal{W}|})$ is the vector of arrival rates of transactions for each class, and $D = [D_{i,r}]$ is a $|\mathcal{R}| \times |\mathcal{W}|$ matrix of service demands. The service demand $D_{i,r}$ for workload class r at resource i is defined as the average total service time spent by transactions of class r at resource i. This does not include any queuing time. Queuing is computed by the Performance Model Solver, which uses well-known techniques for solving open multiclass QN models with load dependent resources [27].

The next section describes how the service demands in the matrix D are computed.

12.1 Computation of Service Demands

A statement s in the specification of a transaction associated with class r in CLISSPE may contribute to the service

demand of various resources. For example, a `select` statement includes CPU demands at the DB server, at the storage box of the database server and at the various networks that connect the application server that issues the `select` statement and the database server. In general, we can write that

$$D_{i,r} = \sum_{s \in S_{i,r}} n_s \times p_s \times D^s_{i,r}, \quad (1)$$

where

- $S_{i,r}$ is the set of all statements that contribute to the service demand of transactions of class r at resource i,
- n_s is the average number of times that statement s is executed,
- p_s is the probability that statement s is executed, and
- $D^s_{i,r}$ is the average service demand at resource i for class r due a single execution of statement s.

Let us first describe how n_s and p_s are obtained. If a statement s is not within any loop statement, then $n_s = 1$. The value of n_s can be modified by the `loop` statement in CLISSPE. This statement has the form `loop` $<$ `number` $>$ $<$ `statement`$_1$ $>; \ldots;$ $<$ `statement`$_m$ $>;$ `end_loop`; where <number> is the average number of times that the sequence of $m (m \geq 1)$ statements is executed. If a loop statement is not nested within any other loop statement, then n_s = `number` for all statements s in the sequence of statements within the loop. In general,

$$n_s = \prod_{j=1}^{K_s} \mathrm{NLoop}_j, \quad (2)$$

where K_s is the number of nested loops in which statement s is part of and NLoop_j is the average number of times loop j is executed.

The probability p_s that a statement s is executed can be modified by three CLISSPE statements: the `if-then`, the `if-then-else`, and the `switch` statements. The `if then` statement is of the form `if` $<$ `prob` $>$ `then` $<$ `statement`$_1$ $>$ $; \ldots;<$ `statement`$_m$ $>;$ `end_if;`, where <prob> is a number in the range $[0, 1]$ that indicates the probability the statements in the `then` clause are executed. If the `if-then` statement is the outermost statement in the transaction, then p_s = <prob> for all statements s in the `then` clause. In general, the probability p_s that a statement s in a `then` clause is executed can be written as $p_s =<$ `prob` $> \times p_{if-then}$, where $p_{if-then}$ is the probability that the `if-then` statement is executed.

The `if-then-else` statement is of the form

```
if <prob> then
   < statement_t1 >;...; < statement_tm > ;
else < statement_e1 >;...;< statement_en > ;
end_if;.
```

The probability p_s that a statement s in the `then` clause is executed can be written as $p_s =<$ `prob` $> \times p_{if-then-else}$, where $p_{if-then-else}$ is the probability that the `if-then-else` statement is executed. The probability p_s that a statement s in the `else` clause is executed can be written as $p_s = (1 - <$ `prob` $>) \times p_{if-then-else}$.

The `switch` statement is of the form

```
switch
      case <prob_1>:
            <statement_11>; ...; <statement_1m>;
      ...;
      case <prob_k>:
            <statement_k1>; ...; <statement_kn>;
end_switch;}
```

where $\sum_{j=1}^{k}$ `prob`$_j = 1$. Then, the probability that a statement s in the jth `case` clause is executed is given by `prob`$_j \times$ p$_{switch}$, where p_{switch} is the probability that the `switch` statement is executed.

One of the major challenges in estimating the service demands $D^s_{i,r}$, is when s is a database statement such as `select`. The next section describes in detail the models used by the CLISSPE compiler to estimate the number of I/Os and the disk time and CPU times associated with each database access.

13 Step 8-b: DB Modeling

This section discusses how the CLISSPE compiler computes the estimated CPU and I/O costs associated with `select` statements. The performance of a database select statement is a function of the number of I/Os generated by the statement. The number of I/Os is a function of the access plan (e.g., nested loop join, merge join, hybrid join) chosen by the query optimizer of the DBMS to perform the select, of the existence of indexes and type of access method (e.g., b-tree, hashing) used in each table, the buffer size and buffer management policies (e.g., LRU), and of parameters such as page sizes, data and index page fill factors, and others. For relevant previous work on access plans, join processing, and query optimization see [2], [5], [29], [37], [38], [42], [44], [45]. Some of these papers describe the operation of access plans and others concentrate on estimating the resulting size of joins between relations, for various types of joins. We build on existing work and present an integrated view that is aimed at computing the total cost, including access to indices, of a given database access.

13.1 Indexing

An index on a database table T is a table with two columns. Each row is of the form (IndexKey, RowPointer), where an IndexKey is either a value found in one or more rows of T for a single column, or a concatenation of column values in a specified order. A RowPointer (called rowid in Oracle, rid in DB2, and tid in Ingres) uniquely identifies a row in T. Rows are stored in specific slots within database pages. Pages are stored within operating system files (more about page formats will be given later).

All commercial DBMSs support indexes based on b-trees. Some, like ORACLE [3], support other methods such as hash clusters and index sequential access methods (isam).

Indexes may be clustered or unclustered. A clustered index is one in which the rows are referenced in the index in the same order as they are stored in the database. Only one

index may be clustered for each table. Thus, when rows with a common index key value are clustered together and one reads in the data page from disk containing one of the rows with a given key value, other rows with the same key value are likely to lie on the same data page. The CLISSPE compiler assumes that if the index is nonclustered, rows with the same key value require one I/O each, while for clustered indexes, the number of I/Os is roughly equal to the number of rows to be read divided by the number of rows per page.

13.1.1 B-Tree Index

The format of a page of a b-tree index is [Header, ($\text{KeyValue}_1, \text{RowPointer}_1), \cdots, (\text{KeyValue}_k, \text{RowPointer}_k)$, FreeSpace]. The height of a b-tree with fanout k and L entries is given by

$$h = \lceil \log_k L \rceil. \tag{3}$$

The fanout k_I for a b-tree index I can be computed as

$$k_I = \left\lfloor \frac{(P - h_i) \times f_i}{ks_I + rp} \right\rfloor, \tag{4}$$

where

- P: size in bytes of an index page (the same as the size of a data page).
- h_i: size in bytes of the header field of an index page.
- f_i: percentage of the useful space of an index page that can be used to store keys and row pointers. In an active b-tree, this value can be shown to be 71 percent on the average for nodes below the root [29].
- ks_I: size in bytes of a key for index I.
- rp: size in bytes of a row pointer.

In most cases of practical interest, the height of the b-tree will be at most three. The root of the tree will always be in memory. Depending on the number of indexes and buffer size, it is quite likely that all level-two blocks will also be in main memory. So, the number of I/Os to access a b-tree index will be one in most cases.

Consider the function $g(T, I)$, shown in Fig. 8, that returns the number of I/Os needed to reach a leaf node for a b-tree structured index I on table T. The term NRows_T represents the number of rows of table T.

13.1.2 Hash Index

In this case, rows are stored at a location pointed by a row pointer determined by applying a hash function to the index key. Hash primary indexes are available in INGRES and ORACLE version 7. The average number of accesses to a hash index for table T, nh_T , is given by

$$nh_T = \frac{1}{1 - (f_d)^{nrp_T}}, \tag{5}$$

where

- nrp_T: number of rows of table T that fit in a database page. This is computed in Subsection 13.2 and
- f_d: percentage of the useful space of a page that can be used to store rows and row directory entries.

```
function g(table T, index I): real;

    fanout = ⌊ ((P − h_i) × f_i) / (ks_I + rp) ⌋ ;
    height = ⌈ log_fanout NRows_T ⌉;

if height < 3
then g = 0
else g = height - 2 ; /* account for buffered index
                         nodes */
end function g;
```

Fig. 8. Function g.

The above result assumes that rehashing is done on the same page. Only when the data page is full, rehashing is done on another page. The derivation of (5) is beyond the scope of this paper, but it is based on deriving the probability distribution of the number of hash misses and then computing the expected value of this distribution [29].

13.2 Data Pages

The format of a data page may vary slightly from DBMS to DBMS. We assume the following format as an adequate abstraction of a page layout: [Header, RowDirectory, FreeSpace, $\text{Row}_N, \cdots, \text{Row}_1$].

The row directory is a table with as many entries as the number of rows in the page. An entry in this directory contains the byte offset of the row within the page. The slot number of a row in a page is the number of its entry in the row directory. To allow for row size expansion, the space on a data page is not fully used. A fill factor, f_d, is assumed that indicates the percentage of the page's useful space used to store rows. Thus, the number nrp_T of rows in a page containing rows for table T can be computed as

$$nrp_T = \left\lfloor \frac{(P - h_d) \times f_d}{(rs_T + rd)} \right\rfloor, \tag{6}$$

where

- P: size in bytes of a data page,
- h_d: size in bytes of the header field,
- rs_T: size in bytes of a row of table T, and
- rd: size in bytes of an entry in the row directory.

Equation 6 assumes that a data page only stores rows of a single table. The number of data pages, nd_T, needed to store all pages of table T is then given by

$$nd_T = \left\lceil \frac{\text{NRows}_T}{nrp_T} \right\rceil. \tag{7}$$

13.3 Access Plans

In estimating the cost of an access plan, we assume that the CPU cost is a linear function of the number of I/Os generated by the access plan. Thus,

$$C_{\text{CPU}}(AcessPlan) = a \times N_{I/O}(AccessPlan) + b, \tag{8}$$

where $N_{I/O}(AccessPlan)$ is the number of I/Os incurred by the access plan and a and b are constants to be determined

by benchmarking the database on a specific environment. The constant b stands for a startup CPU cost and the constant a represents the CPU cost per I/O.

The following types of access plans are considered in the following subsections.

- Table space scan.
- Index scans:
  - single table single index,
  - single table multiple indexes,
  - two table joins (nested loop, merge, and hybrid joins), and
  - more than two table joins.

13.3.1 Table Space Scan (TS)

The simplest access plan is a *table scan (TS)* which is a scan of all rows of a table. This is the preferred method when the number of data pages needed to store all rows of the table is relatively small. In this case, the query optimizer ignores all existing indexes and scans all the rows checking if they match the specified predicates in the select statement. It is also the only possible method when there are no indexes.

The number of I/Os, $N_{I/O}(TS)$, for a table scan of table T is given by the number of data pages needed to store the table. Thus,

$$N_{IO}(TS) = nd_T = \left\lceil \frac{\text{NRows}_T}{nrp_T} \right\rceil. \tag{9}$$

The I/O cost $C_{I/O}$, measured in time units for a table scan on table T is given by

$$C^T_{I/O}(TS) = \left(\left\lceil \frac{\text{NRows}_T}{nrp_T} \right\rceil - 1) \right) \times S^{\text{seq}}_{IO} + S^{\text{rand}}_{IO}, \tag{10}$$

where

- S^{seq}_{IO}: time needed to do a sequential I/O. No seek is needed and the rotational delay is assumed to be one full rotation time. See [21] for a discussion of computation of service times on magnetic disks.
- S^{rand}_{IO}: time needed for a random I/O. It is the sum of a seek time plus half of a rotation time plus the transfer time.

DB2 is able to do *sequential prefetch* at a rate of 32 page reads in sequence saving the rotational delay between successive reads [29]. DB2 also implements *list-prefetch*, where 32, nonnecessarily consecutive, data pages are provided to the disk controller that will optimize access. The following rule-of-thumb is used by the CLISSPE compiler to establish a relationship between random (S^{rand}_{IO}), sequential-prefetch (S^{sp}_{IO}), and list-prefetch (S^{lp}_{IO}) disk service times per I/O.

$$S^{\text{rand}}_{IO} = 10 \times S^{\text{sp}}_{IO} = 2.5 \times S^{\text{lp}}_{IO}. \tag{11}$$

So, for DB2, the formula for the cost of a table scan access plan on table T is

$$C^T_{IO}(TS) = \left(\left\lceil \frac{\text{NRows}_T}{nrp_T} \right\rceil - 1) \right) \times S^{\text{sp}}_{IO} + S^{\text{rand}}_{IO}. \tag{12}$$

The CLISSPE compiler assumes that random I/O is always executed if three or fewer pages are read.

13.3.2 Indexed Scans

This section considers the cost of executing a select statement when one or more indexes are available. The discussion starts with the simplest case of a single table select and then considers multiple table cases.

Single Table Select with a Single Matching Index (STSI). Consider a select statement on table T with predicates on columns $C_1, \cdots, C_m$ as given below

$$\text{select from } T \text{ where } C_1, \cdots, C_m;$$

with a *single matching index* defined as an index where the indexing key is the concatenation of keys $C_1, \cdots, C_k$ for $1 \leq k \leq m$. The number of rows filtered by the index, nrs_T, is computed as

$$nrs_T = \text{NRows}_T \prod_{i=1}^{k} s(C_i), \tag{13}$$

where $s(C_1), \cdots, s(C_k)$ are the selectivity factors for columns $C_1, \cdots, C_k$. The selectivity factor for a given column is the fraction of the total number of rows selected by the predicate.

The access plan consists in following the B-tree index for the concatenated key $(C_1, \cdots, C_k)$ using the proper key values for the predicates in $C_1, \cdots, C_k$ until the leftmost leaf with such a value is found. Then, the leaves of the b-tree are traversed in sequence, following the leaf pointers, while there are entries with the same key value. For each such entry, the row pointed out by the row pointer is retrieved. To compute the number of I/Os involved in executing such select statement, the following definitions are in order:

- I_{ag}: index on the aggregate key $(C_1, \cdots, C_k)$.
- Function NLeaves (NKeys, I) that computes the number of leaf nodes traversed to scan NKeys consecutive keys in index I:

$$\text{NLeaves}(\text{Nkeys}, \text{I}) = \lceil \text{Nkeys}/k_I \rceil. \tag{14}$$

So, the number of I/Os is given by the sum of:

- The number of I/Os needed to go from the root of the B-tree down to the leaf page with the leftmost entry for the appropriate key in the B-tree. This is given by $g(T, I_{\text{ag}})$.
- The number of additional leaf nodes to be traversed in the B-tree. To compute this, we assume that, on the average, the first key of interest is in the middle of the first leaf node found when descending from the root to the leaves. The total number of entries in leaf nodes with the required key value is equal to the number, nrs_T, of rows of table T that satisfy the select statement. Of these entries, $k_{I_{\text{ag}}}/2$ entries are in the first leaf and the remaining $nrs_T - k_{I_{\text{ag}}}/2$ are in additional leaf nodes. Thus, the total number of additional leaf nodes that contain relevant entries is NLeaves $(nrs_T - k_{I_{\text{ag}}}/2, I_{\text{ag}})$.

```
function gstsi (table T, index I, int nrs_T): real
    cost ← g(T, I) × S_IO^rand;
    cost ← cost +NLeaves(nrst_T − k_{I_ag}/2, I_ag) × S_IO^seq;
    if I is clustered
    then cost ← cost +⌈nrs_T/nrp_T⌉ × S_IO^seq
    else cost ← cost +nrs_T × S_IO^rand;
    gstsi ← cost;
end function gstsi;
```

Fig. 9. Function gstsi.

- The number of data pages to be read. If the index is not clustered, we assume that the number of data pages is equal to the number of rows pointed by the index for the given key value. If the index is clustered, the number of data pages read is $\lceil nrs_T/nrp_T \rceil$.

Then, the number $N_{IO}(STSI)$ of I/Os for a STSI access plan can be written as

$$g(T, I_{\text{ag}}) + \text{NLeaves}(nrs_T - k_{I_{\text{ag}}}/2, I_{\text{ag}}) + nrs_T \qquad (15)$$

for nonclustered indexes and as

$$g(T, I_{\text{ag}}) + \text{NLeaves}(nrs_T - k_{I_{\text{ag}}}/2, I_{\text{ag}}) + \lceil nrs_T/nrp_T \rceil \qquad (16)$$

for clustered indexes.

So, the cost $C_{IO}(STSI)$ of the single table select with a single selectable index is given by $C_{IO}(STSI) = gstsi(T, I_{\text{ag}}, nrs_T)$ where the function $gstsi$ is given in Fig. 9.

Single Table Select with Multiple Index Access (STMI). With Multiple Index Access, the query optimizer of the DBMS extracts a list of row pointers from each index. Then, these lists are intersected (for AND predicates) and/or unioned (for OR predicates). The resulting list corresponds to the list of rows that should be retrieved. The row pointers from each index are stored in main memory into a candidate list that is sorted for later sort-merge with the other lists. Since the row pointer lists are processed in memory, there is no I/O cost associated with merging these lists. Since, row pointers to data pages are available in sorted order before the access is made, list prefetch is assumed when retrieving the data pages resulting from the final list.

The purpose of each index is to decrease the number of data pages to be retrieved. Indexes with a very large selectivity factor are not very useful and are avoided by the query optimizer. The query optimizer orders the indexes in increasing order of selectivity factors. Indexes are used from the beginning of the list to the end until the number of row pointers in the list is less than the number of rows that would be selected by the next index. For example, consider a table with 10,000,000 rows, and four indexes I1, I2, I3, and I4 with selectivity factors of 0.0001, 0.01, 0.02, and 0.5. The first index generates a row pointer list of 1,000 elements. The second index generates a list with 100,000 elements. The list generated by the third index has 200,000 elements, and finally the list generated by the fourth index has 5,000,000 elements. The intersection of the first two lists will generate a list with 1,000 x 0.01 = 10 elements. At this point, it is clearly more advantageous to read in the 10 rows and screen them for the two remaining predicates than to scan 200,000 and 5,000,000 entries of index.

```
function gstmi (table T): real;
    totcost ← 0; i ← 0; nrids ← NRows_T;
    repeat
     i ← i + 1;
     /* cost to get to leftmost leaf */
     cost ← g(T, I_i) × S_IO^rand;
     /* add cost to scan index */
     cost ← cost +[NLeaves(L_i, I_i) − 1] × S_IO^seq;
     /* update total cost */
     totalcost ← totalcost + cost;
     /* compute number of rows selected so far */
     nrids ← nrids × s(C_i);
     if i < k
     then if (nrids × S_IO^lp) <
              NLeaves(L_{i+1}, I_{i+1}) × S_IO^seq
          /* exit from loop: don't use other indices */
          i ← k;
    until (i = k);
    /* add cost to retrieve rows using list prefetch */
    gstmi ← totcost + nrids ×S_IO^lp;
end function gstmi;
```

Fig. 10. Function gstmi.

Consider the select statement

$$\text{select from } T \text{ where } C_1, \cdots, C_k, \cdots, C_m;$$

where $C_1, \cdots, C_k$ have indexes with selectivity factors of $s(C_1), \cdots, s(C_k)$, respectively. We assume, without loss of generality, that $s(C_1) < s(C_2) < \cdots < s(C_k)$. Let $L_1, \cdots, L_k$ denote the length of the row pointer lists generated by the indexes on $C_1, \cdots, C_k$, respectively. Thus,

$$L_i = s(C_i) \times \text{NRows}_{\text{T}}$$

for $i = 1, \cdots, k$.

The I/O cost for a STMI access plan is then given by $C_{IO}^T(STMI) = gstmi(T)$ where the function $gstmi$, shown in Fig. 10, computes the IO cost for a select statement with a single table and multiple indexes.

The number of I/Os, $N_{IO}(STMI)$, generated by a select executed using STMI is given by the function nstmi given in Fig. 11.

Multiple Table (Join) Select Statements. We consider first two table selects before we discuss multiple table ones. A join involving tables T_1 and T_2 is written in CLISSPE as

$$\begin{array}{lll} \text{select from} & \text{where} & C_1^1, \cdots, C_m^1 \\ \text{from} & \text{where} & C_1^2, \cdots, C_n^2 \\ & \text{joined\_by} & T_1.C_j^1 = T_2.C_k^2; \end{array}$$

where C_j^1 and C_k^2 are the *joining columns* and are not part of the predicate lists $C_1^1, \cdots, C_m^1$ and $C_1^2, \cdots, C_n^2$, respectively.

This section considers nested loop join, merge scan join, and hybrid join access plans.

Nested Loop Joins. One of the two tables being joined is called the outer table, the one from where rows are retrieved first and the second is called the inner table. For

```
function nstmi (table T): real;
    totIO ← 0; i ← 0; nrids ← NRows_T;
    repeat
      i ← i + 1;
      /* no. of IOs to get to leftmost leaf in index */
      numIOs ← g(T, I_i);
      /* add I/Os to scan index */
      numIOs ← numIOs + (NLeaves(L_i, I_i) − 1);
      totIO ← totIO + numIO;
      /* compute number of rows selected so far */
      if i < k
      then if (nrids × S_IO^lp) <
              NLeaves(L_{i+1}, I_{i+1}) × S_IO^seq
           /* exit from loop: don't use other indices */
           i ← k;
    until (i = k);
    /* add IOs to retrieve rows */
    nstmi ← totIO + nrids;
end function nstmi;
```

Fig. 11. Function nstmi.

each qualified row in the outer table, all qualified and joined rows of the inner table are retrieved. For now, let T_1 be the outer table and T_2 be the inner table. We discuss later, in more detail, how the query optimizer decides which table should be the outer table. The general procedure for executing a join using the nest loop join method is:

1. If there are any indexes for $C_1^1, \cdots, C_m^1$, then use an index-based access plan to determine a list L of row pointers for all rows in T_1 that satisfy the indexable predicates. If there are no indexes, then the list L is null.
2. All rows in L are retrieved and checked to see if they qualify for remaining predicates in $C_1^1, \cdots, C_m^1$ not considered by the index. The retrieval of rows in this case can be made using list prefetch since the row pointers are known in advance. If the list L is null, then a table scan is performed on T_1 to determine all rows that qualify for $C_1^1, \cdots, C_m^1$. Let the set of rows of T_1 obtained by this step be called the q-outer table (for qualified outer table) denoted by T_1^q.
3. For each row in T_1^q find all rows in T_2 that have a value in the joining column C_k^2 matching the value in the joining column of T_1, C_j^1. If T_2 has an index for C_k^2, then an indexed access method can be used to retrieve the rows in T_2 with a fixed value for C_k^2. If there is no index in T_2 for C_k^2, then all rows of T_2 have to be retrieved and checked to see if they qualify for the joining column as well as for any other predicates in $C_1^2, \cdots, C_n^2$.

Then, the total cost of a nested loop join with two tables can be written as

$$C_{IO}(\text{NestedLoop}) = C_{IO}(T_1) + \text{NumberQualRowsT}_1 \times C_{IO}(T_2). \quad (17)$$

The number of rows NumberQualRowsT_1 of T_1 that are qualified according to the predicates $C_1^1, \cdots, C_m^1$ is given by

TABLE 1
Query Optimizer Decision for Nested Loop Joins

Index on Join Column		Other Index		Outer Table	Access Plan	
T_1	T_2	T_1	T_2		T_1	T_2
No	No	No	No	max $\mid T \mid$	TS	TS
No	No	No	Yes	T_1	TS	X_o
No	No	Yes	No	T_2	X_o	TS
No	No	Yes	Yes	min X_o	X_o	X_o
No	Yes	No	No	T_1	TS	X_j
No	Yes	No	Yes	T_1	TS	X_j
No	Yes	Yes	No	T_1	X_o	X_j
No	Yes	Yes	Yes	T_1	X_o	X_j
Yes	No	No	No	T_2	X_j	TS
Yes	No	No	Yes	T_2	X_j	X_o
Yes	No	Yes	No	T_2	X_j	TS
Yes	No	Yes	Yes	T_2	X_j	X_o
Yes	Yes	No	No	min X_j	X_j	X_j
Yes	Yes	No	Yes	min X_j	X_j	X_j
Yes	Yes	Yes	No	min X_j	X_j	X_j
Yes	Yes	Yes	Yes	min X_j	X_j	X_j

$$\text{NumberQualRowsT}_1 = \text{NRows}_{T_1} \times \prod_{i=1}^{m} s(C_i^1). \quad (18)$$

Thus, we can write that

$$C_{IO}(\text{NestedLoop}) = C_{IO}(T_1) + \left(\text{NRows}_{T_1} \times \prod_{i=1}^{m} s(C_i^1)\right) \times C_{IO}(T_2). \quad (19)$$

The costs $C_{IO}(T_1)$ and $C_{IO}(T_2)$ are computed as a basis of the access plan used by the query optimizer to retrieve rows of T_1 and T_2. These decisions are explained in what follows.

A nested loop join is more efficient if the joining column for the inner table has an index on it and a small number of rows from the outer table qualify for the join. The decisions made by the query optimizer are summarized in Table 1. The first two columns indicate whether or not there is an index on the joining column for tables T_1 and T_2. The third and fourth columns indicate whether there is an index on a column other than the joining column. The fifth column indicates the table that is selected as the outer table. Finally, the last two columns indicate the access plans used to retrieve rows from tables T_1 and T_2. The following notation was used in Table 1:

- max $\mid T \mid$: table with the largest number of rows,
- X_o: single table select with a matching index on a column other than the joining column,
- X_j: single table select with a matching index on the joining column,
- min X_o: table with the smallest number of rows selected through index X_o,
- min X_j: table with the smallest number of rows selected through index X_j, and
- TS: table scan.

If the inner table is small enough so that its index and data pages fit into the buffer after the first time they are referenced, then, the IO cost becomes zero when these pages are retrieved after the first time.

Merge Join (MJ). Also known as merge scan join or sort merge join, merge join scans tables T_1 and T_2 only once in the order of their join columns. The general strategy for a merge join can be summarized as follows:

1. Execute a select statement of the form `select from` T_1 `where` $C_1^1, \cdots, C_m^1$ and retrieve the resulting rows sorted by the joining column C_j^1 into a temporary table denoted by T_1^t. The number of rows, $\mathrm{NRows}_{T_1^t}$, of this temporary table is computed as

$$\mathrm{NRows}_{T_1^t} = nrs_{T_1} = \mathrm{NRows}_{T_1} \times \prod_{i=1}^{m} s(C_i^1). \tag{20}$$

2. Execute a select statement of the form `select from` T_2 `where` $C_1^2, \cdots, C_n^2$ and retrieve the resulting rows sorted by the joining column C_k^2 into a temporary table denoted by T_2^t. The number of rows, $\mathrm{NRows}_{T_2^t}$, of this temporary table is computed as

$$\mathrm{NRows}_{T_2^t} = nrs_{T_2} = \mathrm{NRows}_{T_2} \times \prod_{i=1}^{n} s(C_i^2). \tag{21}$$

3. Scan tables T_1^t and T_2^t once using a merge procedure to find the rows in each table that have matching joining column values. If the number of rows $\mathrm{NRows}_{T_1^t}$ and $\mathrm{NRows}_{T_2^t}$ in tables T_1^t and T_2^t is small, then the merge step is performed in memory and there is no I/O cost associated. Otherwise, these rows have to be read from disk. Random read is assumed in this case.

Let us define the function

$$\mathrm{InMem}(\mathrm{Bytes}, \mathrm{BufferSize}) = \begin{cases} 1 & \mathrm{Bytes} \le \mathrm{BufferSize} \\ 0 & \mathrm{Bytes} > \mathrm{BufferSize} \end{cases} \tag{22}$$

to help in indicating which of the temporary tables are stored in buffers for use in the merge step.

The total I/O cost $C_{IO}(MJ)$ of a merge join can then be written as,

$$\begin{aligned} C_{IO}(MJ) \; = & \; C_{IO}(\mathrm{select}_1) + C_{IO}(\mathrm{select}_2) + C_{IO}(\mathrm{sort}_1) + \\ & C_{IO}(\mathrm{sort}_2) + C_{IO}(\mathrm{merge}), \end{aligned} \tag{23}$$

where the cost of the selects on tables T_1 and T_2 depends on the existence of indexes on the predicates indicated in the select statement. These costs were already computed in the previous subsections.

The merge cost $C_{IO}(\mathrm{merge})$ is computed as follows. Let $\mathrm{LT}_i (i = 1, 2)$ be the size in bytes of the temporary table T_i^t computed as $nrs_{T_i^t} \times rs_{Ti}$. Then,

$$\begin{aligned} C_{IO}(\mathrm{merge}) = & S_{IO}^{\mathrm{rand}}[(1 - \mathrm{InMem}(\mathrm{LT}_1, BS)) \times \mathrm{LT}_1 + \\ & (1 - \mathrm{InMem}(\mathrm{LT}_1 + \mathrm{LT}_2, BS)) \times \mathrm{LT}_2], \end{aligned} \tag{24}$$

where BS is the buffer size, in bytes, of the DBMS. The expression above assumes that the first temporary table has priority over the second table in using the buffer.

The number of I/Os for executing a disk sort using an M-way sort algorithm on D data pages is given by $2 \times D \times \lceil \log_M D \rceil$ (see [29]). So, the sort cost of steps 1 and 2 are computed as follows: The number of data pages D_1 in temporary table T_1^t is

$$D_1 = \left\lceil \frac{\mathrm{NRows}_{T_1^t}}{nrp_{T_1}} \right\rceil. \tag{25}$$

Let SB be the size in bytes of the buffer area used by the DBMS for performing sorts and other auxiliary functions. Then, to execute an M-way sort, we need M+1 pages in the buffer. With a sort buffer of SB bytes, one can store $\lfloor SB/P \rfloor$ pages. So, $M = \lfloor SB/P \rfloor - 1$. Finally,

$$C_{IO}(\mathrm{sort}_1) = 2 \times D_1 \times \lceil \log_M D_1 \rceil \times S_{IO}^{\mathrm{rand}}. \tag{26}$$

Using similar arguments we have that

$$D_2 = \left\lceil \frac{\mathrm{NRows}_{T_2^t}}{nrp_{T_2}} \right\rceil \tag{27}$$

and

$$C_{IO}(\mathrm{sort}_2) = 2 \times D_2 \times \lceil \log_M D_2 \rceil \times S_{IO}^{\mathrm{rand}}. \tag{28}$$

Hybrid Join (HJ). This type of join is executed as follows:

1. As in merge join, execute a select statement of the form `select from` T_1 `where` $C_1^1, \cdots, C_m^1$ and retrieve the resulting rows sorted by the joining column C_j^1 into a temporary table denoted by T_1^t. The number of rows, $\mathrm{NRows}_{T_1^t}$, of this temporary table is given by (20).
2. Scan the temporary table T_1^t, in joining column order. For each value of the joining column, perform an index lookup in table T_2 and retrieve the row pointers for the rows in T_2 that satisfy this lookup. Let I_2 be the index on table T_2 for the joining column C_k^2. Another temporary table, T_2^t , is built containing rows with the columns of T_1^t plus an additional column for the row pointer of the qualifying table T_2 rows. The number of rows, $\mathrm{NRows}_{T_2^t}$, in table T_2^t is given by $\mathrm{NRows}_{T_1^t} \times \; s(C_k^2) \times \mathrm{NRows}(T_2)$. The cost, $C_{IO}(\mathrm{rid})$, of retrieving the row pointers in this step is given by (29). The term in square brackets indicates the I/O cost incurred to do an index lookup for each value of the joining column in the outer temporary table. The number of rows in the temporary outer table if one eliminates duplicate values in the joining column is obtained by applying the selectivity factor of the joining column to the outer temporary table.

$$\begin{aligned} C_{IO}(\mathrm{rid}) = & [g(T_2, I_{C_k^2}) \times S_{IO}^{\mathrm{rand}} + \\ & (\mathrm{NLeaves}(s(C_k^2) \times \mathrm{NRows}_{T_2}, I_{C_k^2}) - 1) \times \\ & S_{IO}^{\mathrm{seq}}] \times \\ & \mathrm{NRows}_{T_1^t} \times s(C_j^1). \end{aligned} \tag{29}$$

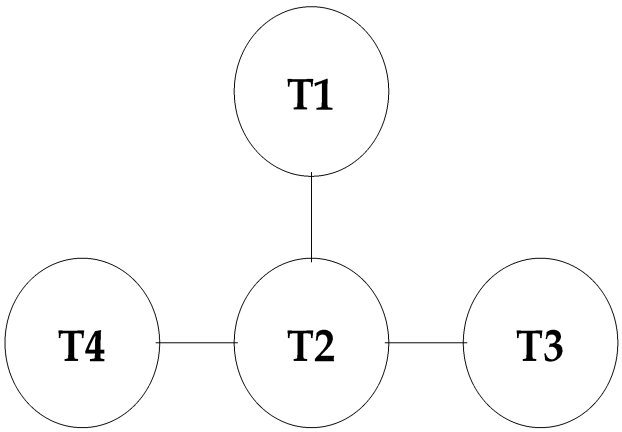

Fig. 12. Example of a join graph.

Following the same arguments presented in the Merge Join section, the cost of performing step 1 for a hybrid join is given by $C_{IO}(\text{select}_1) + C_{IO}(\text{sort}_1)$.

So, the cost $C_{IO}(HJ)$ of a hybrid join is given by

$$C_{IO}(HJ) = C_{IO}(\text{select}_1) + C_{IO}(\text{sort}_1) + C_{IO}(\text{rid}). \quad (30)$$

More than Two Table Joins. When more than two tables are joined, the query optimizer has to decide the order in which joins are performed. Given a specific order, tables are joined pairwise, using the methods described in the previous subsections, and the temporary table generated is joined with the other tables. For example, in the three-table join below, tables T_1, T_2, and T_3 are joined. The query optimizer may decide to join T_1 and T_2 to generate a temporary table T_t which is then joined to T_3, or join T_2 and T_3 first to generate T_t which is then joined to T_1.

```
select  from    T1 where  C1^1, ..., Cm^1
        from    T2 where  C1^2, ..., Cn^2
        from    T3 where  C1^3, ..., Cp^3
            joined_by  T1.Cj^1 = T2.Ck^2,
                       T2.Ci^2 = T3.Ck^3;
```

In general, the number of possible join order alternatives may be quite large and the computational effort of the query optimizer may be nontrivial if one takes into account all possible join methods that can be used for each join. The approach taken by the CLISSPE compiler is to consider join order alternatives only. If n tables are being joined, the number of possible join orders is less than or equal to $n!/2$ and greater than or equal to $n-1$. Thus, if a select statement joins four tables, there are at most 12 alternatives to consider. For five tables there are at most 60. The actual number of alternative join orders depends on the joins specified in the `joined_by` clause. CLISSPE, performs an exhaustive search of all possible join orders since it is not expected that CLISSPE programs will have select statements that join more than five tables.

The CLISSPE compiler uses the following algorithm to compute the cost, $C_{IO}(MTJ)$, of a multitable join:

1. Build a list of join alternative (LJA) orders taking into account the joins specified in the `joined_by` clause.
2. For each element $a = (T_1, \cdots, T_n)$ in the list LJA, compute the I/O cost $C_{IO}(a)$ of the alternative a as

$$C_{IO}(a) = C_{IO}(T_1 \bowtie T_2) + C_{IO}((T_1 \bowtie T_2) \bowtie T_3) + \cdots + \quad C_{IO}((T_1 \bowtie \cdots T_{n-1}) \bowtie T_n), \quad (31)$$

   where $\bowtie$ stands for the join operator. The I/O cost computed for each individual join is obtained by selecting the join method (i.e., nested loop, merge join, and hybrid join) that provides the minimal cost for that join. From (31), it is clear that one needs to be able to derive all the needed parameters for a table resulting from a join so that one can compute the cost of joining this table with the next one in the list. This implies that one needs to compute the size $SJ(T_i, T_j)$ of a table resulting from the join of tables T_i and T_j. The derivation of $SJ(T_i, T_j)$ is given below after the algorithm to compute the list LJA is presented.
3. The cost $C_{IO}(MTJ)$ is computed as the minimum cost among all costs $C_{IO}(a)$ for all elements in the list LJA.

To present the algorithm to build the list LJA, some definitions are in order. Let $G = (V, E)$ be a nondirected join graph where the vertices are the tables of a select statement. An edge $e = (T_i, T_j) \in E$ indicates that there is a join between tables T_i and T_j. Fig. 12 shows an example of a join graph for a select statement with the following joins: $T_1 \bowtie T_2$, $T_2 \bowtie T_3$, and $T_2 \bowtie T_4$. The list LJA for the graph of Fig. 12 is (T_1, T_2, T_3, T_4), (T_1, T_2, T_4, T_3), (T_2, T_3, T_1, T_4), (T_2, T_3, T_4, T_1), (T_2, T_4, T_1, T_3), and (T_2, T_4, T_3, T_1). Note that since $T_i \bowtie T_j = T_j \bowtie T_i \quad \forall\, i, j$, all elements in the list LJA that start with T_i, T_j are equivalent to elements that start with T_j, T_i and, therefore, must not be duplicated. Let $\Pi = [T_{i_1}, \cdots, T_{i_m}]$ be a sequence of nodes in V and let the operation $\Pi \leftarrow \Pi \oplus T$ indicate that node T is added at the end of the sequence Π resulting in the sequence $\Pi = [T_{i_1}, \cdots, T_{i_m}, T]$. Let L_j denote a set of node sequences of length j. Let V_Π be the set of nodes included in the sequence Π and let E_Π be the set of edges in E that connect a node in V_Π to a node in $V - V_\Pi$.

The algorithm to build the list LJA for a graph $G = (V, E)$ is given in Fig. 13. Let $| V | = n$. The algorithm of Fig. 13 works by building sequences of length 2 up to n tables. Sequences of length j are built out of sequences of length $(j-1)$ by adding at the end of the sequence a node not yet in the sequence but that is connected to any node in the sequence. This means that the added node represents a table that can be joined with the tables already represented in the sequence.

The computation of the size $SJ(T_i, T_j)$ of a table resulting from the join of tables T_i and T_j on columns C_i and C_j is as follows. Let $m = 1/s(C_i)$ be the number of different values in column C_i and $n = 1/s(C_j)$ be the number of different values in column C_j. Assume for the moment that $m > n$. Let $k = s(C_j) \times \text{NRows}_{T_j}$ be the number of rows of T_j that

```
LJA ← ∅;
For each e = (T_i, T_j) ∈ E do
  begin
    j ← 2; L_j ← {[T_i, T_j]};
    while j < n do
      begin
        j ← j + 1; L_j ← ∅;
        For each Π ∈ L_{j-1} do
          For each v ∈ (V − V_Π) s.t.
                    ∃u ∈ V_Π ∧ (v, u) ∈ E_Π do
                      L_j ← L_j ⋃ {Π ⊕ v};
      end;
    LJA ← LJA ⋃ L_n;
  end
```

Fig. 13. Algorithm to build LJA.

have the same value in column C_j. Each row in table T_i will not have any row with the value of C_i equal to the value of C_j with probability P_0 equal to

$$P_0 = \frac{\binom{m-1}{n}}{\binom{m}{n}} = \frac{m-n}{m}. \tag{32}$$

Each row of table T_i will generate k rows in the result relation with probability $(1 - P_0)$. Thus, the average number of rows generated in the result relation per row of T_i is given by

$$\begin{aligned} k \times (1 - P_0) = k \times \frac{n}{m} = \\ s(C_j) \times \text{NRows}_{T_j} \times \frac{1/s(C_j)}{1/s(C_i)} = \\ \text{NRows}_{T_j} \times s(C_i). \end{aligned} \tag{33}$$

Then, $SJ(T_i, T_j)$ can be computed as,

$$SJ(T_i, T_j) = \text{NRows}_{T_i} \times \text{NRows}_{T_j} \times s(C_i). \tag{34}$$

In general, relaxing the assumption that $m > n$, we can write that

$$SJ(T_i, T_j) = \text{NRows}_{T_i} \times \text{NRows}_{T_j} \times \min(s(C_i), s(C_j)). \tag{35}$$

If there are other predicates in the `from` clauses for tables T_i and T_j, one has to multiply $SJ(T_i, T_j)$ by the product of the selectivity factor for all predicates involved.

13.3.3 Access Plan Selection in CLISSPE

The CLISSPE compiler selects the proper access plan to be used in estimating the I/O and CPU costs of a select query by computing the cost of all possible access plans and selecting the one with the minimum cost. The type of DBMS specified in the CLISSPE program may restrict the types of access plans considered by the CLISSPE compiler and may also restrict the types of disk I/Os to be used. For example, since list prefetch is restricted to DB2, the CLISSPE compiler replaces S_{IO}^{lp} by S_{IO}^{rand} for all DBMSs other than DB2.

14 Step 8-c: Performance Parameter Gathering

In a performance specification, there are many performance parameters that need to be estimated. At the design level, when the application is not operational, these parameters cannot be obtained by performance measurement. To characterize the workload intensity, the expected frequency of execution of each use case is estimated. In the case study application (Section 5), it is the number of times a new applicant is processed and the number of times an existing employee is processed over some time period, e.g., per day. For each use case, the number of transactions is estimated (some transactions may be executed more than once per use case). If the application is a reengineering of an existing application, then the workload, based on use cases and transactions per use case, can be measured. The existing transaction load is then used as a baseline transaction load indicated by an arrival rate multiplier of 1.0. Experiments can be run with larger transaction loads by increasing the arrival rate multiplier.

Other application-level parameters also need to be estimated. The transaction specifications given in Section 10 illustrate examples of such parameters, which are used in CLISSPE branch and loop statements: #ProbabilityApplicantWithMinSkills, #average_skill_count, #average_num_courses, #avg_sections_count, and #ProbSectionAvailable. These parameter values can be determined from the legacy system if it collects such data. If not, then, providing historical data is kept in the files or database maintained by the legacy application, programs can be written to analyze the records and determine the parameters. If there is no computerized information available, then interviews with key users are required. Users can provide these estimates based on their first-hand experience or by manual analysis of the paper files. In the application we modeled, a combination of all the above was used to determine the values of the parameters. In total, 65 constants and 35 probabilities needed to be estimated. To estimate the size of the database, estimates of row size were determined by adding the estimated size of each attribute of the relation. Estimates of table size were obtained by a combination of analysis of the existing system and interviews with users. As the legacy system used a file management system and the client/server system was designed to use a relational database, a direct comparison of the two systems was not possible. The main performance challenge for the client/server system was not the size of the database but the expected high transaction rate.

15 Step 8-d: Performance Assessment

As an example of applying the technology, this section describes the performance analysis of the Recruitment and Training System of a major US Government Agency, in which the current system is being downsized and re-engineered from a mainframe to a client/server environment using a relational database. Some results of this analysis were reported in [19].

TABLE 2
Average Response Time for Check New Applicant Transaction

	Load Multiplier			
	1.0	1.5	2.0	2.5
Arrival Rate (tps)	0.047	0.071	0.095	0.12
Scenario	Response Time (sec)			
1D(2)1A(1)	46.7	83.2	4243	N/A
1D(2)1A(1)SC	15.7	22.8	3436	N/A
3D(2)1A(1)	30.8	37.0	48.2	76.4
3D(2)2A(1)SC	9.5	10.1	10.9	12.3
1D(4)1A(2)SC	8.0	8.4	9.1	10.3
1D(3)1A(1)SC	8.0	8.5	9.2	10.4
Comb(6)SC	5.8	5.8	5.8	5.9
Comb(4)SC	5.8	5.8	5.8	5.9

Several alternative client/server scenarios were modeled and analyzed, eight of which are reported in this section. They fall into three main categories: centralized (i.e., one database server machine and one application server machine), distributed (more than one database server and/or application server machines), and combined (a single machine supporting the database and application servers). Scenario names are of two forms: *mD(p)nA(q)* or *Comb(p)*. In the former case, m is the number of database servers, p the number of processors in each database server machine, n is the number of application servers, and q the number of processors in each machine supporting the application server. In the $Comb(p)$ notation, p indicates the number of processors on the machine used to execute the database and application servers. The suffix SC, if present in the scenario name, indicates that the scenario uses partial caching with Selective table full Caching. This means that some of the tables are fully cached in main memory. For example, the scenario 3D(2)2A(1)SC indicates that three dual-processor machines are used as database servers and two single-processor machines are used as application servers. Also, full caching of selected entire tables is used. All machines used as database servers were assumed to be high-end UNIX servers. For the first four scenarios in Table 2, the machines used as DB servers are 67 percent slower than the ones in the last four scenarios. Except for scenario 3D(2)2A(1)SC, which uses an NT server on a Wintel platform, all other scenarios use a UNIX server to support the application server. In all scenarios, the clients used Wintel NT platforms.

The workload on the existing mainframe system was measured and the transaction load on the new system was estimated as described in Section 14. The transaction load was run at arrival rate multipliers ranging from 1.0 to 2.5. A rate multiplier of 1.0 corresponds to the peak transaction load observed in the current system. The arrival rate multipliers were used to verify how the new system would react to a predicted load increase. With the help of the CLISSPE system we were able to analyze the system response time for the critical transactions. Table 2 shows the average response times for the Check New Applicant transaction for the different client/server configurations and rate multipliers. The Check New Applicant transaction is the most critical and also the most demanding transaction, which used most of the use cases shown in Fig. 3: Check New Applicant Qualifications, Check Skills, Check Training Opportunities, Check Incentives, and Check Job Location. The complexity of the transaction is considerably higher than the examples given in the earlier sections of this paper.

As can be seen in Table 2, the average response times for scenarios 1D(2)1A(1), 1D(2)1A(1)SC, and 3D(2)1A(1) is inadequate. The response times for the remaining five scenarios are satisfactory at transaction arrival rate multipliers of up to 2.5 times the current transaction load.

We now compare the scenario 3D(2)2A(1)SC with scenarios 1D(4)1A(2)SC and 1D(3)1A(1)SC. The distributed scenario 3D(2)2A(1)SC has three dual-processor database server machines for a total of six processors. Response times for this scenario are around 20 percent higher than for the centralized scenarios 1D(4)1A(2)SC and 1D(3)1A(1)SC which have a single database server machine with four and three processors, respectively. This is mainly due to the fact that the processors in the 3D(2)2A(1)SC scenario are 67 percent slower than those in the 1D(4)1A(2)SC and 1D(3)1A(1)SC ones.

The CLISSPE system also allows one to determine the resources in which each transactions spends most of its time. For the centralized and distributed configurations 3D(2)2A(1)SC, 1D(4)1A(2)SC, and 1D(3)1A(1)SC, the limiting resource is the LAN that connects the Application Server(s) to the Database Server(S). Approximately 43 percent of the average response time is spent in the LAN in the two centralized scenarios and 36 percent in the distributed scenario.

In the two combined scenarios, (Comb(6)SC and Comb(4)SC), there is no LAN connecting the database and application servers. These two scenarios provide the lower average response time among all eight scenarios. This is due to the elimination of the LAN which in the previous scenarios is used every time the database is accessed. It should be pointed out that this is a rather intensive database application. The limiting resource in the combined scenarios becomes the CPU at the combined server machine. Transaction Check New Applicant spends around 52 percent of its average response time at the processors. It should also be pointed out that significant caching is assumed here. The servers were configured with sufficient main memory to allow for the caching of the critical tables. While caching reduces the amount of I/O, it does not reduce the amount of processing associated with the execution of database accesses.

The results for the combined scenarios are virtually identical for rate multipliers of up to 2.5. This indicated that there is adequate spare CPU capacity in the six-processor configuration. Note that this application cannot benefit from parallelism. Having more processors only reduces waiting time for a processor but does not reduce processing time. In fact, after modeling scenario Comb(6)SC with six CPUs, we decided to model a less powerful configuration scenario Comb(4)SC with four CPUs. This indicates that the four-processor configuration is adequate. It can also be seen

that there is very little contention in the combined configurations. The response time is almost flat in the range of load multipliers going from 1.0 to 2.5.

Due to space constraints, we cannot elaborate on the analysis of all major transactions and the various tradeoffs. However, we give a brief idea of the major recommendations. Two configurations were selected as being acceptable: 1D(3)1A(1)SC and Comb(4)SC. Both use high-end UNIX servers for the database and application server machines. The CPUs in each machine are of the faster type. The first scenario provides a higher degree of reliability than the second since it can be reconfigured as a combined scenario with somewhat degraded performance if one machine goes down.

In both cases, sufficient main memory was recommended to cache critical database tables. One of the outputs of the CLISSPE system is the number of I/Os executed by each `select` statement as well as the average number of times it is executed. This allowed for transaction redesign to improve performance and for the determination of the tables to be cached. In fact, the analysis revealed that, without full caching of the most critical tables (over 100Mbytes in total), a satisfactory level of performance could not be achieved with the configurations modeled.

The design and performance analysis carried out for the RTS system was very thorough. It consisted of roughly 10,000 lines of CLISSPE code and provided application programmers with detailed guidelines on how to implement the transactions. The analysis also provided a specification of the system configuration to be procured by the agency.

16 Discussion and Future Work

In the current version of the system, the mapping from the object collaboration diagram (and its documentation in the message sequence descriptions) to the CLISSPE transaction specification is done by hand. Initially, the transaction specification was specified in two procedures, one for the client and one for the application server. The client specification corresponds to the user interface object shown in Figs. 5 and 6. However, the server objects, e.g., the coordinator and entity objects shown in Figs. 5 and 6, are mapped to the server procedure.

The CLISSPE specification of each transaction was developed from the use case and object collaboration diagram for the transaction. At the pseudocode level, the logic of the transaction, in terms of the relations accessed and processing required for each business rule, was reviewed with the users and developers. Based on early results of the performance analysis, the transactions with the greatest resource demands were determined and analyzed in more detail. Alternative algorithms were considered involving a different order for executing the business rules and, hence, a different access pattern to the database relations. Each alternative algorithm was modeled and compared with other candidate algorithms. Using this approach, significant performance improvements were obtained in key algorithms.

As we applied the initial version of CLISSPE to the real-world Recruiting and Training System, it became apparent that for long transactions, having one nonmodular server procedure, resulted in lengthy procedures, which did not exploit the benefits of the use case and object structuring carried out in the OO analysis and design. To address this problem, a macro capability with parameters was introduced into CLISSPE, which allows a long transaction specification to be decomposed into smaller macros, which correspond to the objects and/or operations depicted in the object collaboration diagrams. This meant that macros could be reused in different transaction specifications just as objects are reused in different use cases and object collaborations.

In the future, we are planning to provide a closer link between the UML model for use cases and object collaboration diagrams and the CLISSPE specification. Thus, each class would have a corresponding CLISSPE specification. An object collaboration diagram for a transaction would be mapped to a transaction specification by creating instances of the classes and binding the instances into a transaction specification. We are also investigating using a component based model of a distributed software architecture [6], [18], specified in an Architecture Description Language [17], [39], with the specification of each component to include a description of its performance characteristics.

Another issue is that in changing from one client/server configuration to a different one, some amount of recoding needs to be done. Although the main change is in the mapping section of the specification, other changes are also needed in the transaction specification, in particular where it specifies which node the transaction executes on. With the ADL approach outlined above, the configuration of the application could be separated from the transaction specifications, making it easier to specify and experiment with different configurations.

17 Concluding Remarks

Software design and development in a C/S environment offers a very large number of design alternatives to the designer. These alternatives range from software architectural issues to the choice of proper system configuration and platforms to be used. Predicting the performance of a system under development requires that service demands be estimated for resources such as CPU, disks, and networks.

For I/O intensive applications, a good estimate of the CPU time of a transaction can be obtained as a function of the number of I/Os. The challenge then is to estimate the number of I/Os. Most SPE studies and packages require the software performance engineer to provide these estimates by analyzing the transaction logic. This may lead to quite inaccurate estimates. In this paper, we modeled the work performed by the query optimizer of the DBMS in order to generate more accurate estimates on the number of I/Os for different query types and different types of indices on the tables being queried. The models we developed are based on analytic formulas that capture the traversal of indices and processing of various types of joins. In our studies, we were able to see clearly the effects of changing database configurations. For example, adding or removing indices

from critical relations had a major impact on the number of I/Os.

Our understanding of the operation of the query optimizer for commercial databases such as Oracle and DB2 was limited to what is known in the published literature. Proprietary aspects of the query optimizers were not modeled. Therefore, discrepancies between the model estimates and actual measurements may arise. However, in SPE at the early stage of the system development, one is interested in relative performance and not necessarily in absolute performance values.

Our work shows that these early performance models, which are developed before the system exists, have an important role to play. These models may be used for comparative studies of alternative hardware and software architectures, even though they are not expected to be as accurate as more detailed models that use measurements from a real system. As the system is developed, it is possible to refine the model by including measurements from the real system.

SPE requires the collaboration of the software developers who are usually too busy to get the system running on schedule. Performance is usually an afterthought. This paper presents a method that blends software design with performance modeling. The method is based on the CLISSPE language that can be used to specify use cases and also as a basis for generating predictive performance models and its parameters.

CLISSPE can be used by both software system designers and performance engineers. One of the major deterrents for the widespread use of SPE is that it is viewed by many as an activity separate from software design and development, and, therefore, should be carried out by people with different skills. With our integrated method, we hope to bridge the gap between these two camps.

ACKNOWLEDGMENTS

This work was partially supported by the US National Science Foundation under grant CCR-9804113. The authors would like to thank the anonymous reviewers for their thoughtful suggestions which helped improve the quality of the paper.

REFERENCES

[1] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide.* Reading, Mass.: Addison Wesley, 1999.

[2] S. Christodoulakis, "Estimating Block Transfers and Join Sizes," *Proc. ACM 1983 SIGMOD Conf.,* pp. 40-54, May, 1983.

[3] G. Franks and M. Woodside, "Performance of Multi-Level Client-Server Systems with Parallel Service Operations," *Proc. First Int'l Workshop Software and Performance,* Oct. 1998.

[4] G. Franks, A. Hubbard, S. Majumdar, D. Petriu, J. Rolia, and C.M. Woodside, "A Toolset for Performance Engineering and Software Design of Client-Server Systems," *Performance Evaluation J.,* vol. 24, no. 1-2, pp. 117-135, Nov. 1995.

[5] D. Gardy and C. Puech, "On the Effect of Join Operations on Relation Sizes," *ACM Transactions on Database Systems,* vol. 14, no. 4, pp. 574-603, Dec. 1989.

[6] H. Gomaa and G. Farrukh, "Composition of Software Architectures from Reusable Architecture Patterns," *Proc. IEEE Int'l Workshop Software Architectures,* Nov. 1998.

[7] H. Gomaa, "Use Cases for Distributed Real-Time Software Architectures," *J. Parallel and Distributed Computing Practices,* June 1998.

[8] H. Gomaa and G. Farrukh, "Automated Configuration of Distributed Applications from Reusable Software Architectures," *Proc. IEEE Int'l Conf. Automated Software Engineering,* Nov. 1997.

[9] H. Gomaa, D.A. Menascé, and L. Kerschberg, "A Software Architectural Design Method for Large-Scale Distributed Data Intensive Information Systems," *Journal of Distributed Systems Engineering,* vol. 3, pp. 162-172, 1996.

[10] H. Gomaa, *Designing Concurrent, Distributed, and Real-Time Applications with UML.* Addison Wesley, 1993.

[11] A Grummitt, "A Performance Engineer's View of System Development and Trials," *Proc. 1991 Computer Measurement Group Conf.,* pp. 455-463, Dec. 1991.

[12] D. Harel, "On Visual Formalisms," *Comm. ACM,* vol. 31, no. 5, pp. 514-530, May 1988.

[13] H. Hlavacs and G. Kotsis, "Modeling User Behavior: A Layered Approach," *Proc. Seventh Int'l Symp. Modeling,* Oct. 1999.

[14] G. Jacobson, G. Booch, J. Rumbaugh, *The Unified Software Development Process.* Reading, Mass.: Addison Wesley, 1999.

[15] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, *Object-Oriented Software Engineering.* Reading, Mass.: Addison Wesley, 1992.

[16] G. Koch and K. Loney, *Oracle: The Complete Reference Electronic Edition.* Oracle Press, 1996.

[17] D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, and W. Mann, "Specification and Analysis of System Architecture using Rapide," *IEEE Trans. Software Eng.,* vol. 21, no. 4, Apr. 1995.

[18] J. Magee, N. Dulay, and J. Kramer, "Regis: A Constructive Development Environment for Parallel and Distributed Programs," *J. Distributed Systems Eng.,* pp. 304-312, 1994.

[19] D.A. Menascé and H. Gomaa, "On a Language Based Method for Software Performance Engineering of Client/Server Systems," *Proc. First Int'l Workshop Software and Performance,* Oct. 1998.

[20] D.A. Menascé and V.A.F. Almeida, "Performance of Client/Server Systems," *Performance Evaluation—Origins and Directions,* G. Haring, C. Lindemann, and M. Reiser, eds. Springer-Verlag, 2000.

[21] D.A. Menascé and V.A.F. Almeida, *"Capacity Planning for Web Performance: Metrics, Methods, and Models."* Upper Saddle River, N.J.: Prentice Hall, 1998.

[22] D.A. Menascé, "A Framework for Software Performance Engineering of Client/Server Systems," *Proc. 1997 Computer Measurement Group Conf.,* Dec. 1997.

[23] D.A. Menascé, "CLISSPE: A Language for Client/Server Software Performance Engineering," technical report, Dept. of Computer Science, George Mason Univ., Jan. 1997, http://www.cs.gmu.edu/~menasce/clisspe.

[24] D.A. Menascé, V. Almeida, R. Fonseca, and M.A. Mendes, "A Methodology for Workload Characterization of E-Commerce Sites," *Proc. ACM Conf. Electronic Commerce,* Nov. 1999

[25] D.A. Menascé, O. Pentakalos, and Y. Yesha, "An Analytic Model of Hierarchical Mass Storage Systems with Network-Attached Storage Devices," *Proc. ACM Sigmetrics Conf.,* May 1996.

[26] D.A. Menascé, H. Gomaa, and L. Kerschberg, "A Performance-Oriented Design Methodology for Large-Scale Distributed Data Intensive Information Systems," *Proc. First IEEE Int'l Conf. Eng. of Complex Computer Systems,* Nov. 1995.

[27] D.A. Menascé, V.A.F. Almeida, and L.W. Dowdy, *Capacity Planning and Performance Modeling: from Mainframes to Client-Server Systems.* Upper Saddle River, N.J.: Prentice Hall, 1994.

[28] J.E. Neilson, C.M. Woodside, D.C. Petriu, and S. Majumdar, "Software Bottlenecking in Client-Server Systems and Rendezvous Networks," *IEEE Trans. Software Eng.,* vol. 21, no. 9, pp. 776-782, Sept. 1995.

[29] P. O'Neil, *Database Principles, Programming, Performance.* San Francisco: Morgan Kauffman, 1994.

[30] D. Parnas, "Designing Software for Ease of Extension and Contraction," *IEEE Trans. Software Eng.,* Mar. 1979.

[31] D. Parnas, "On the Criteria for Decomposing a System into Modules," *Comm. ACM,* Dec. 1972.

[32] S. Ramesh and H.G. Perros, "A Multi-Layer Client-Server Queuing Network Model with Synchronous and Asynchronous Messages," *Proc. First Int'l Workshop Software and Performance,* Oct. 1998.

[33] M. Reiser and S. Lavenberg, "Mean-Value Analysis of Closed Multi-Chain Queuing Networks," *J. ACM,* vol. 27, no. 2, 1980.
[34] J.A. Rolia and K.C. Sevcik, "The Method of Layers," *IEEE Trans. Software Eng.,* vol. 21, no. 8, pp. 689-700, Aug. 1995.
[35] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual.* Reading, Mass.: Addison Wesley, 1999.
[36] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-oriented Modeling and Design.* Upper Saddle River, N.J.: Prentice Hall, 1991.
[37] S. Salza and M. Terranova, "Evaluating the Size of Queries on Relational Databases with Non-Uniform Distribution and Stochastic Dependence," *Proc. ACM SIGMOD Conf.,* pp. 8-14, June 1989.
[38] L. Shapiro, "Join Processing in Database Systems with Large Main Memories," *ACM Trans. Database Systems,* vol. 11, no. 3, pp. 239-264, Sept. 1986.
[39] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996
[40] C. Smith, P. Clements, and M. Woodside, *Proc. First Int'l Workshop Software and Performance,* Oct. 1998.
[41] C. Smith, *Performance Engineering of Software Systems.* Reading, Mass.: Addison Wesley, 1990.
[42] A. Swami and K.B. Schiefer, "Estimating Page Fetches for Index Scans with Finite LRU Buffers," *Proc. ACM SIGMOD Conference,* pp. 173-184, 1994.
[43] C.M. Woodside, J.E. Neilson, D.C. Petriu, and S. Majumdar, "The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software," *IEEE Trans. Computers,* vol. 44, no. 1, Jan. 1995.
[44] S.B. Yao, "Optimization of Query Evaluation Algorithms," *ACM Trans. Database Systems,* vol. 4, no. 2, pp. 133-155 Sept. 1979.
[45] C.T. Yu, "Distributed Database Query Processing," *Query Processing in Database Systems,* W. Kim, D. Reiner and D. Batory, eds. Springer-Verlag, 1985.
[46] M. Woodside, C. Hrischuck, B. Selic, and S. Bayarov, "A Wideband Approach to Integrating Performance Prediction into a Software Design Environment," *Proc. First Int'l Workshop Software and Performance,* Oct. 1998.

Daniel A. Menascé received the PhD degree in computer science from UCLA (1978), the MSc degree in computer science, and the BSEE degree both from the Pontifical Catholic University in Rio de Janeiro (PUC-RIO), Brazil (1975 and 1974, respectively). He is a professor of computer science at George Mason University, Fairfax, Virginia. He held visiting faculty positions at UMIACS, University of Maryland at College Park (1991-1992), and at the University of Rome, Italy (1983). He was a full time faculty member of the Department of Computer Science at PUC-RIO, Brazil, for 14 years, where he was also chair of CS (1981-1983). Menascé is an ACM Fellow and a member of IFIP's Working Group 7.3. He has published more than 110 technical papers and was the chief author of five books, including *Scaling for E-Business: Technologies, Models, Performance, and Capacity Planning, Capacity Planning for Web Performance: Metrics, Models, and Methods,* and *Capacity Planning and Performance Modeling: From Mainframes to Client-Server Systems,* published by Prentice Hall in 2000, 1998, and 1994, respectively. His research has been funded by DARPA, NASA, NSF, Virginia's Center for Innovative Technology, OPNET Technologies, Hughes Applied Information Systems, Brazilian Telecommunications Company, Brazilian Research Council (CNPq), Brazilian Ministry of Science and Technology, and IBM Brazil. Menascé was the recipient of various prizes, teaching awards, and best paper awards. Menascé was general chair of ACM Sigmetrics 1999 and program co-chair of the 2000 ACM Workshop on Software Performance. His areas of interest include performance modelling, web and e-commerce technologies, and software performance engineering. He is a member of the IEEE Computer Society.

Hassan Gomaa received the BSc degree with first class honors in electrical engineering from University College, London University, and the DIC and PhD degrees in computer science from Imperial College of Science and Technology, London University. He is a professor in the Department of Information and Software Engineering and associate director of the Center for Information Systems Integration and Evolution at George Mason University, Fairfax, Virginia. He has more than 25 years experience in software engineering, both in industry and academia, and has published more than 100 technical papers. His book, *Software Design Methods for Concurrent and Real-Time Systems,* was published by Addison Wesley as part of the SEI Series on Software Engineering and had its fourth printing in 1999. His new book entitled *Designing Concurrent, Distributed, and Real-Time Applications with UML,* was also published by Addison Wesley in August 2000. His current research interests include object-oriented analysis and design for concurrent, real-time, and distributed systems, software architecture, domain analysis and design, software reuse, software performance engineering, intelligent software agents, client/server systems, software engineering environments, software prototyping, and software process models. A recent paper he co-authored with D.A. Menasce on the design of large-scale information intensive client/server systems received an outstanding paper award at the IEEE Internationall Conference on the Engineering of Complex Computer Systems in November, 1995. He has served on the program committees of several international conferences and was program co-chair of the 1997 IEEE International Conference on the Engineering of Complex Computer Systems and was program co-chair of the 2000 ACM Workshop on Software Performance. He is a member of the IEEE Computer Society.