Hierarchical Modeling of Large-Scale Systems Using Relational Databases

by

Ting-Sheng Fu

A Thesis Submitted to the Faculty of the
DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements for the Degree of
MASTER OF SCIENCE
In the Graduate College
The University of Arizona

2002

STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for an advanced degree at the University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the copyright holder.

SIGNED: ______________________________

APPROVAL BY THESIS DIRECTORS

This thesis has been approved on the date shown below:

______________________________ ______________

Hessam S. Sarjoughian, Ast. Prof. of Computer Science & Engr. Date
Arizona State University

______________________________ ______________

Bernard P. Zeigler, Prof. of Electrical & Computer Engr. Date
University of Arizona

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

LIST OF FIGURES

Acknowledgement

I would like to thank my co-advisors Hessam Sarjoughian and Bernard Zeigler. I also would like to express my appreciation for Prof. Jerzy Rozenblit who served on my defense committee.

This research has been supported in part by NSF Next Generation Software (NGS) grant #EIA-997505 and NSF Manufacture Enterprise Systems grant #DMI-0075557.

Abstract

Modeling of large-scale systems plays a major role during analysis and design phases of many software/system developments. To aid modeling of a large number of model components, it is important to employ large-scale repositories Relational Database Management Systems that provide systematic and efficient centralized medium for storing and accessing models. In this thesis, we describe our approach to analysis, design, and implementation of Scaleable System Entity Structure Modeler (SESM). The modeling environment implements a variant of System Entity Structure (SES) formulism using client-server architecture. We discuss our formulation of an Entity Relation schema (based on Relational Database constructs) that captures hierarchical system structures using concepts of modularity, input/output ports, and coupling among model components. The schema represents a mapping from object-oriented modeling constructs to their relation-oriented counterparts. We describe our development of the SESM environment using the Java programming enterprise computing technologies and Oracle-8i Relational Database Management Systems (RDBMS). The client-side of SESM environment provides graphical user interface for model creation, retrieval, and manipulation. Finally, we also discuss previously developed modeling environments that are closely related to our work.

1. INTRODUCTION

There is a growing need for supporting modeling of large-scale systems. A variety of approaches exist to help system analyst, architects, and designers to specify a system in terms of its structure and its components. These approaches do not generally follow a methodical approach and consequently provide weak support for usability, scalability, and modifiability, and storage of models. An approach providing such capabilities is becoming increasingly important as systems grow in size and complexity. For example, the Boeing 777 aircraft is composed of more than one million parts that had to be engineered – i.e., analyzed, designed, constructed, and tested. Given the scope of models and their roles in engineering processes, systematic and scaleable model development approach plays a central role in use and reuse of models during the lifetime of a large-scale system.

1.1. Increasing Need for Model Repositories

The ability to model is highly dependent on having a repository to support creation, modification, storage, and reuse of models and their structures. Repositories such as a relational database management system provide suitable environment for storing and manipulating model components. Standardized tools (e.g., Oracle8) offer a host of capabilities and services that can be used for modeling systems – i.e., modular, hierarchical structure and relationships among their components. For example, a modular, hierarchical representation of a Switch Network can capture various aspects via a collection of modeling elements. Decomposition (part-of relationship), multiple views/configurations (is-a relationship), as well as other key modeling constructs such as input/output ports, component names and identifications, and attributes of components are key modeling elements. The ability to systematically capture a system in terms of modeling elements in a scaleable and user-friendly repository, therefore, plays a major role for nearly all systems – even systems that contains a few tens of components. Another key related role for model repository is the need for simulation. The models stored in a database can be extended to contain dynamics and therefore simulateable. The services provided by a database directly affects the tremendous growth in simulation and its role in a key technology to the development of many system, particularly, distributed, large-scale systems with hundreds and thousands of components (subsystems).

1.2. Scaleable System Entity Structure Modeler Overview

We first present an overview of the Scaleable System Entity Structure Modeler (SESM). It allows the user to create hierarchical "object-like" models using a relational database and a user interface. Models can be stored and maintained by a relational database management system (RDBMS) and the graphical user interface provides a friendly menu for creating, modifying, and manipulating hierarchical modular models. The SESM is based on the System Entity Structure (SES) formalism [Zei 84; Zei 90]. Every models can be considered as an object having input and output ports. The Scaleable System Entity Structure Modeler (SESM) can represent atomic and hierarchical coupled models that are closely related to the DEVS [Zei 00] models. An atomic model in SHMS has well-defined interfaces (i.e., every component has input and output ports) that are necessary for its interactions with any other model. The coupled models are the same as the DEVS coupled models [Zei 00] in terms of its internal and external couplings. The SESM as presented in this thesis only allows capturing "structural" representation of hierarchical models as defined by DEVS – i.e., dynamics features of atomic models cannot be captured in the SESM. Nevertheless, the underpinning of the approach supports representation of the dynamics of atomic models. SESM may also be extended to support collaborative (multi-user) modeling via a collaborative middleware layer such as Collaborative Distributed Network System [Sun98; Sar00].

To support large-scale modeling, these models must be characterized for representation in a generic relational database such as Oracle. The SESM is composed of a database management system (RDBMS), a network environment, a modeling engine (Server), and a user-interface (Client) as shown in Figure 1. The model data is stored in a relational database. The Server can initialize and manipulate the database based on users requests. The Client displays models for the user and enables modifying them. Both the Client and the Server are connected to the DBMS via a network environment. Server and Client independently initialize and maintain their connectivity. This separation allows user interactions requiring "write access" to the DBMS to be mediated by the Server while "read-access" interactions to be carried out directly with the database.

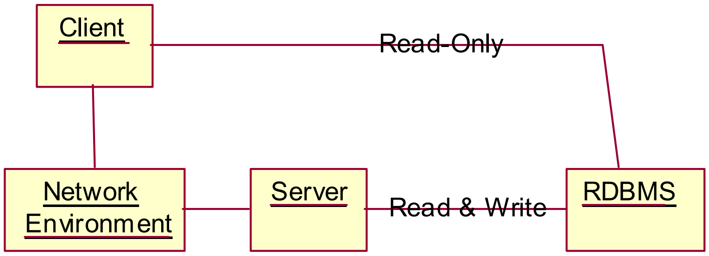


Figure 1: Scaleable System Entity Structure Modeler

1.3. Outline of Thesis

The thesis is organized as follows. In Chapter 2, we describe two closely related work on model representation using systems point of view (how a system can be decomposed into subsystems and the kinds of relationships that may exist among its various parts) and the Unified Modeling Language (UML) which is based on the object-orientation point of view.   In Chapter 3, an overview of databases is presented with emphasis on Relation Database Management System. In this chapter, we also motivate the selection of RDBMS by examining advantages and disadvantages of relational, object, and relational/object database management systems.
The modeling approach is discussed in Chapter 4. We describe the modeling view and elements and discuss their mappings using relational database constructs. In chapters 5 and 6, the design and implementation of the client and server part of the SESM environment is presented. In Chapter 8, we present a user-interface that supports modeling of hierarchical, modular systems. Finally, we discuss conclusions and future work in Chapter 9.

2. SYSTEM DESIGN REPRESENTATION APPROACHES

2.1. System Entity Structure

The System Entity Structure (SES) formalism represents structures of a system in a structural knowledge representation schema [Roz 83, Zei 84, Roz 86]. The formalism allows systematically organizing and representing alternative structures of a system. A system structure is presented as a labeled tree. Each node of the tree is either an entity or an aspect with variable types attached to each node. An example of the 'Switch Network' in SES is shown in the Figure 2.

2.1.1. SES Definition

The fundamental object in the SES formalism is an entity, also known as a model. A model represents a physical object in the real world that is modeled. The model can appear more than once in the entity structure. Each model has identification and a set of variables specifying model structure attached to it. A variable has a name and a range set. The name should be distinct from names of other variables in the same model. The range set is the set of values that the variable can assume. The variables are attached to every appearance of the model as well. For instance, a Packet Switch can be represented as a model. Its identification is 'Packet Switch'. Variables like speed, network ports, and power consumption are contained within the 'Packet Switch' entity. Every appearance of the 'Packet Switch' in the SES tree has the same attached variables.

2.1.2. SES Relationships

The SES also provides three relationships, aspect, decomposition, and specialization. Using these relationships, users can build models hierarchically. The aspects of a model represent alternative decompositions of a system or its model. Thus, the elements of an aspect are models, the components of such decomposition. Similar to an entity, an aspect also has variables. The variables attached to an aspect are variables of that model 's superior model that pertain to it only in the context of the aspect. As seen in the Switch Network example, a switch network can be decomposed in a user aspect or network switch aspect. In the user aspect, a switch network is decomposed into different users, like network administrators, web programmers, terminal users, etc. In the user aspect, the main purpose is to represent the usages of the Switch Network. In the wiring aspect, the Switch Network is decomposed into several Packet switches. The main purpose in this aspect is to represent the connections between the Packet switches within the Switch Network.

The decomposition allows a model (parent), a coupled entity, to be decomposed into one or more component entities (children). The component entities can be of the same label or different labels. When more than one component entities have the same label, the composition is known as the multiple-decomposition. The multiple-decomposition also allows the user to specify a non-fixed number of occurrences of the same label. In the wiring aspect of the Switch Network model, the switch network is decomposed into several Packet Switches.

The specialization relationship allows users to specialize a specialized model (general) with one or more specialization models (specialization). Similar to the specialization in the Object-Orientated Software Engineering, a specialization model must inherit all of the variables from its general model. Other than the variables inherited, a specialization model also has its unique variables labeled by the node. In SES, the specialization relationship is represented by the double arrow as illustrated Switch Network SES Tree Diagram. The Packet Switch entity is specialized by the IP 5 Switch and the IP 6 Switch. Another example is the Network Users. The users can either be on campus (on-campus) or at a remote location (remote).

2.1.3. SES Axioms

In order to build the model systemically, the SES enforces the following axioms [Zei 84].

- Uniformity: Any two nodes that have the same labels are identical. Since nodes with the same labels are the same model, they should have identical attached variable types and isomorphic sub-trees. For instance, the model "Packet Switch" should contain the same variable types, speed, network ports, and power consumption, wherever the node "Packet Switch" appears in the system, "Switch Network".
- Strict Hierarchy: No label appears more than once down any path of the tree. If any label appears more than one in the tree, there must be a loop and it should not be allowed. If the model, "Packet Switch", were further decomposed into models, "Switch Network", it would violate the Strict Hierarchy axiom. The "Switch Network" system became a tree with infinite levels because of the loop between the "Switch Network" and "Packet Switch".

- Alternating Mode: Each node has a mode, which is either "entity" or "specialization"; the mode of a node and the mode of its successors are always opposite. The mode of the root is always entity.
- Valid Brothers: No two brothers have same label. Since each model is identified by its label, no two different models can have the same label.
- Attached Variables: No two variable types attached to the same item have the same name.

Each axiom must be obeyed within the system in order to avoid errors stopping the system to be represented. For instance, if two different models with different attached variable types and sub-trees were both given the identification, "PacketSwitch", a user would have difficulties to identify the correct "PacketSwitch" in the system. Thus, the system is represented ambiguously.

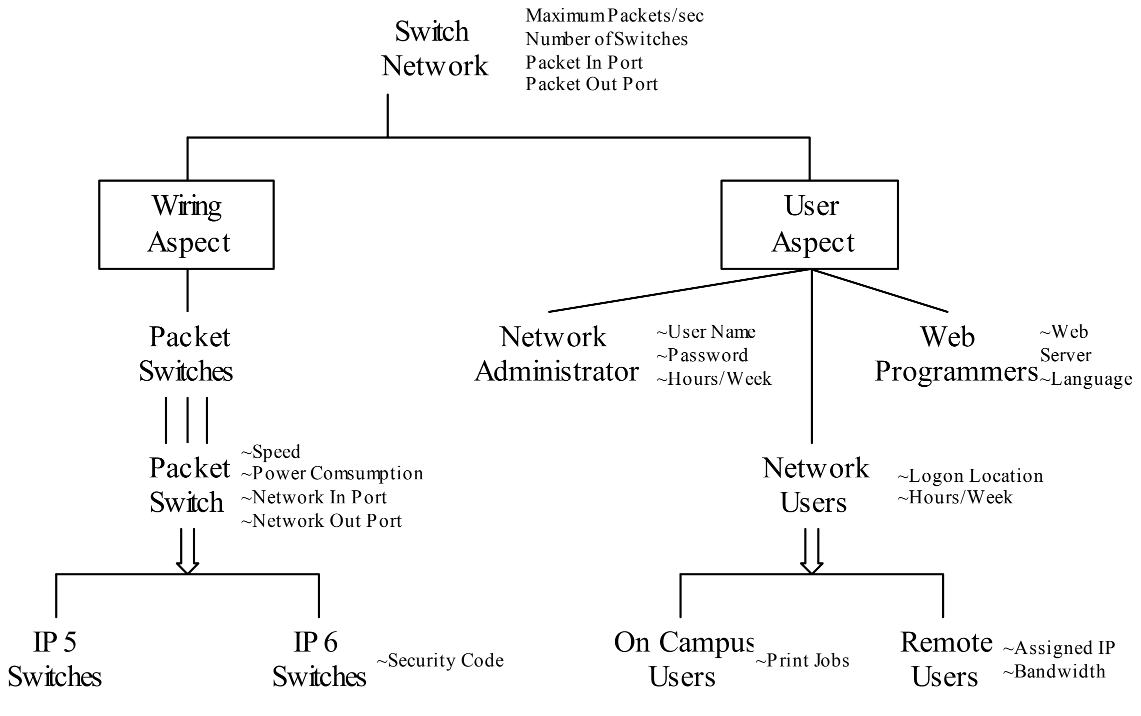


Figure 2: Switch Network SES Tree

2.2. Unified Modeling Language

Another alternative for representing models systematically is by applying the Unified Modeling Language [Fow 00] or its counterparts. The Unified Modeling Language (UML) and SES in some ways are closed related to one another in their representation of a system (or model) structure (i.e., components and their relationships). There are, however, significant differences between them. Next we compare these two approaches.

2.2.1. Similarities and Difference between UML and SES

In UML, each object has attributes that serve the same purpose as variables of a model in SES. Furthermore, UML has relationships like specialization, aggregation, and reference, allowing the user to build system hierarchically [Boo 94; Boo 00; Fow 00]. The specialization in UML maps to the specialization in SES directly. In both representations, the specialization is the "is-a" relationship. The aggregation in UML is very much the same as decomposition in SES. Both aggregation and decomposition represent the "part-of" relationship. Finally, all SES axioms, except the Alternating Mode, are enforced in UML. The Switch Network is also represented in UML as shown in the Switch Network Object Diagram.

Although, the UML and the SES formalism have a lot similarity, there are some significant differences between these two representations. First, UML was designed for general object-oriented software engineering (e.g., [Fow 00]). UML has many features that are not supported in the SES formalism. For instance, the access right of an object's attribute which can be either private, protected or public is not supported in the SES formalism. As a result, UML, not only can be used to represent model structure, it can also be used to represent software structure in general. On the other hand, the SES formalism is specialized in representing models and their structures in a system. Thus, the SES has several characteristics that are difficult to be mapped to UML without further restrictions or modification of UML. For instance, coupling between models is not directly defined in UML. The users have to define the coupling either through an association of the port objects or defining a 'coupling' object. The aspect relationship is another difference between the SES formalism and the UML. While UML allows specifying aspects, a formal definition is not given beside "is-a" relationship. Thus, each aspect of the system must be represented separately as shown in the Switch Network UML Diagram (see Figure 3 and Figure 4).

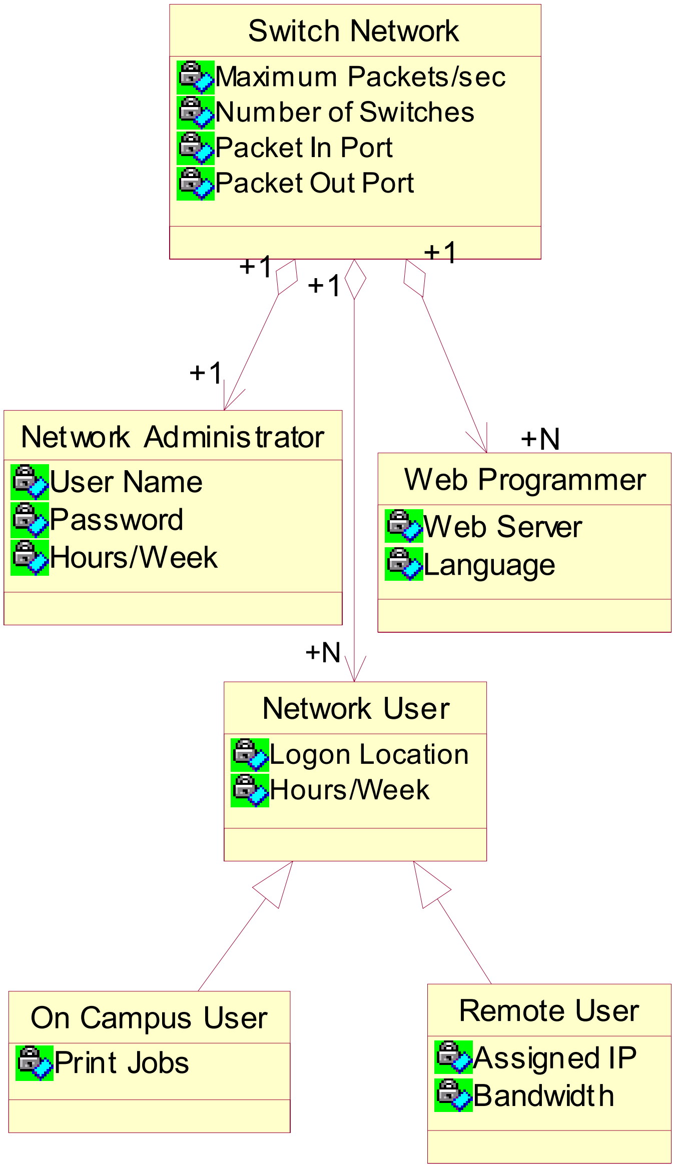


Figure 3: Switch Network UML (User Aspect)

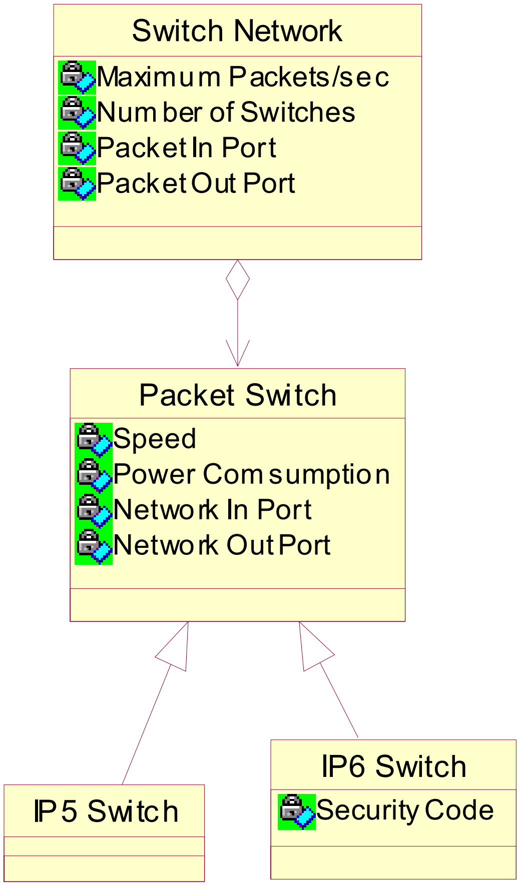


Figure 4: Switch Network UML (Wiring Aspect)

2.3. Relational Algebraic System Entity Structure (RASES)

The RASES is based on both the SES formalism and the Relational Algebra formalism [Par 94]. In other words, RASES represents the SES formalism using relational model and its operations in relational algebra.

2.3.1. RASES Definition

RASES represents SES as a schema based on the relational model. The RASES transforms SES into a relational schema of six tables, RASES = <ASP, SPEC, EVAR, ACOUP, SSEL, GSEL> [Par 97]. These tables are described in details below.

ASP		
ent	asp	subent

Table ASP represents the aspect relationship between entities. Its schema are ent: entity, asp: aspect, and subent (sub-entity): entity belongs to the aspect.

SPEC		
ent	spec	specent

Table SPEC represents the specialization relationship between entities. Its schema are ent(entity): specialized entity, spec (specialization): the label of a particular specialization, and specent (specialization-entity): entity specializes the specialized entity.

EVAR		
ent	variable	value

Table EVAR represents the relationship between entities and attached variables. Its schema are ent: entity, variable: variable, and value: value of the variable.

ACOUP				
asp	ent1	port1	ent2	port2

Table ACOUP represents couplings between entities. Its schema are asp: aspect, ent1: entity, port1: port of ent1, ent2: entity, and port2: port of ent2.

SSEL		
spec	cond	specent

Table SSEL represents the selection rules attached to specializations. Its schema are spec: specialization, con: condition, and specent: specialization entity to be selected if condition is satisfied.

GSEL			
spec1	specent1	spec2	specent2

Table GSEL represents the global selection constraints. Global selection constraints force the selection of spec2 to be specent2 if specent1 is selected as entity for spec1. Its schema are spec1: specialization, specent1: specialization entity, spec2: specialization, and specent2: specialization entity.

2.3.2. RASES Operations

Based on this relational schema, the operations in the SES are defined in the relational algebra. Overall, RASES provides an integrated approach where users can systematically construct simulation models since RASES inherits restrictions, like consistencies among design components, from the SES formalism [Par 96]. These restrictions allows the user create models logically from an abstract level. RASES also allows users to store models in a relational database since it is based on the relational model. With the power of relational database system, users can manage large amount of data, share data among users, and perform fast queries [Par 96]. Thus, RASES is a robust method for model base management.

2.3.3. RASESF Example

The Switch Network example is expressed in RASES below.

ASP		
ent	asp	subent
Switch Network	Wiring	Packet Switch1
Switch Network	Wiring	Packet Switch2
Switch Network	User	Network Administrator
Switch Network	User	Network User
Switch Network	User	Web Programmer
…..		

SPEC		
ent	spec	specent
Packet Switch	PS Spec	IP 5 Switch
Packet Switch	PS Spec	IP 6 Switch
Network User	User Spec	On Campus User
…..		

EVAR		
ent	variable	value
Switch Network	Maximum Packets/sec	600
Switch Network	Number of Switches	5
Switch Network	Packet In Port	port open
Switch Network	Packet Out Port	port open
Packet Switch	Speed	1000
Packet Switch	Power Consumption	5watt/hour
Packet Switch	Network In Port	port open
Packet Switch	Network Out Port	port open
…..		

ACOUP				
asp	ent1	port1	ent2	port2
Wiring	Switch Network	Packet In Port	Packet Switch1	Network In Port
Wiring	Packet Switch1	Network Out Port	Packet Switch2	Network In Port
…..				

SSEL		
spec	cond	specent
User Spec	Logon Location = ‘UA Campus’	On Campus User
…..		

GSEL			
spec1	specent1	spec2	specent2
Network User	Remote User	Packet Switch	IP 5 Switch
Network User	On Campus User	Packet Switch	IP 6 Switch
…..			

2.3.4. Differences Between RASES and SESM

Although the RASES and the SESM representation are both SES representations based on relational model, the two representations are fundamentally different. In the SESM representation, many relationships of the SES, decomposition, ports and couplings, are implemented using relationship directly supported by the relational database. Thus, manipulation of the model (e.g., removing components, adding couplings, etc.) are carried out and maintained by the relational database. RASES, however, depends solely on the implementation of the relational algebra operations to create and maintain the models [Par 94]. Another main difference between the RASES and SESM representation lies in the parts of the SES formalism each can support. The SESM representation supports a more complete representation of decomposition compared to the RASES. In the RASES, multiple decomposition is not represented in the relational algebra. On the other hand, the RASES implements the aspect relationship in SES which is not supported by SESM representation. The RASES also implements variables of an entity. SESM representation supports components, their ports and couplings of models.

3. RATIONALE FOR USE OF DATABASES

3.1. Database System

3.1.1. DBMS & Database

The combination of a database and its management system (DBMS) is often known as a database system [Elm 94]. The relationship between a DBMS, a database and a database system is shown in Figure 5. The database systems have become a very important part of modern information system for the following reasons. A database system provides data independence that allows the separation of applications and physical data storage [O'Ne 01]. With data independence, physical data storage can be scaled without changing the developed application. Furthermore, applications can be developed and enhanced without affecting the data storage. Users can also setup constraints in the database system to ensure the integrity of the data. Moreover, the database system provides utilities for data management and protection that simplifies the application development [Lon 00]. Finally, modern commercial database systems are optimized to ensure performance and scalability. Therefore, the database system is ideal to be used as a central data depository supports multiple users.

A database is set of related data. The data represent some aspects of the real world, the MiniWorld or the Universe of Discourse (UoD) [Elm 94]. In other words, a database contains data that is a logically coherent collection with some inherent meaning. The database is designed, built, and populated with data for a specific purpose with a number of users and applications. Other than data describing the miniworld, database also store meta-data. The meta-data is a complete definition of the database structure and constraints. On the other hand, a database management system (DBMS) is a software system that controls access to the databases and allows users to define, construct and maintain the database. DBMS also maintains the consistency of the data in the database based on the meta-data defined in the database. Commercial DBMSs often provide utilities to assist the users to manage and protect the database [Lon 00]. For instance, most DBMSs have utilities to backup and restore their databases. These utilities further simplify the process of managing the data and improve the availability of the data. A DBMS usually handles several databases and many users.

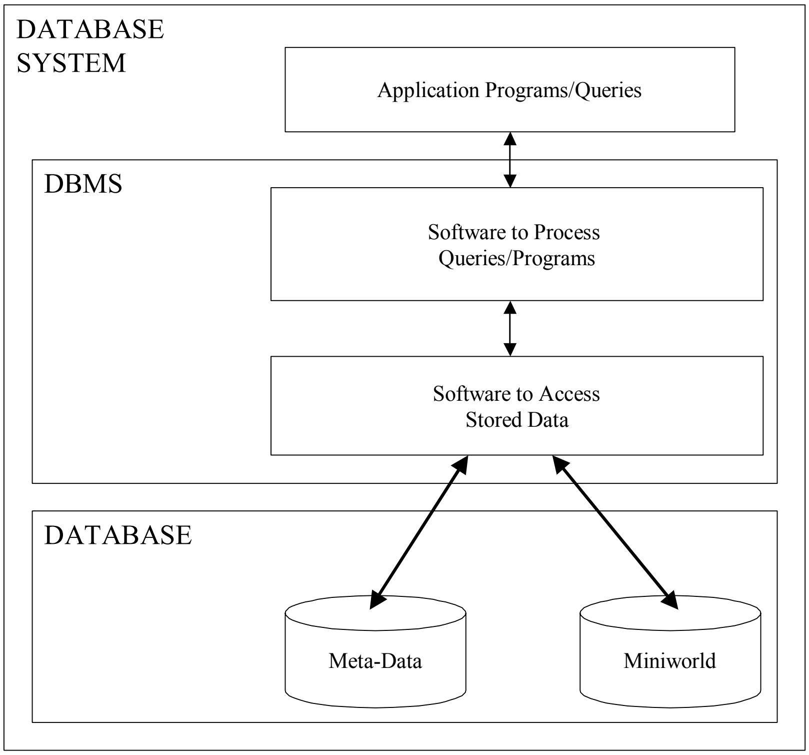


Figure 5: Database & DBMS [Elm 94]

3.1.2. Database System Architecture and Data Independence

The database system uses the three-schema architecture to separate the user applications and the physical database [O'Ne 00]. In the three-schema architecture, schemas can be defined at three level: internal level, conceptual level and external or view level. At the internal level, the schema describes the complete details of physical data storage and access paths for the database. The next level, conceptual level, describes the structure of the whole database for a community of users. The conceptual schema hides the details of physical storages and describes entities, data types, relationships, user operations and constraints. The highest level is the external or view level where external schemas and user views are defined. Due to the complexity of mapping between levels, many DBMSs do not support the external level completely [Bla 98]. The three-schema architecture enables the data independence in database system. With data independence, users of database system can change a level of schema without changing any other levels of schema. This allows different users to operate on the database for different purposes without affecting each other. For instance, a database administrator (DBA) can change the physical data storage from a heap file to a sorted file or add a B-Tree index to improve performance without changing the conceptual schema of the database. As a result, the application implemented based on the conceptual schema will still function correctly although the physical data storage is changed.

3.1.3. DBMS Languages

The three-schema architecture and data independence allows the users of database system to define different levels of schema to support the functions needed. Users define and access the database system using several types of languages. The storage definition language (SDL) is used to specify the internal schema. The data definition language (DDL) is used to define the conceptual schema. The view definition language (VDL) is used to specify user views. Depending on the DBMS, one or all the languages are supported. After the database is defined, the data manipulation language (DML) is used to retrieve, insert, delete, and modify the data. Depending on the different purposes of the user, one or more languages are used by a type of users more often than other languages. Users, like DBA or database designers, often use languages like DDL. On the other hand, an application programmer will use DML primarily. An overview of the users and the languages used is shown in Figure 6.

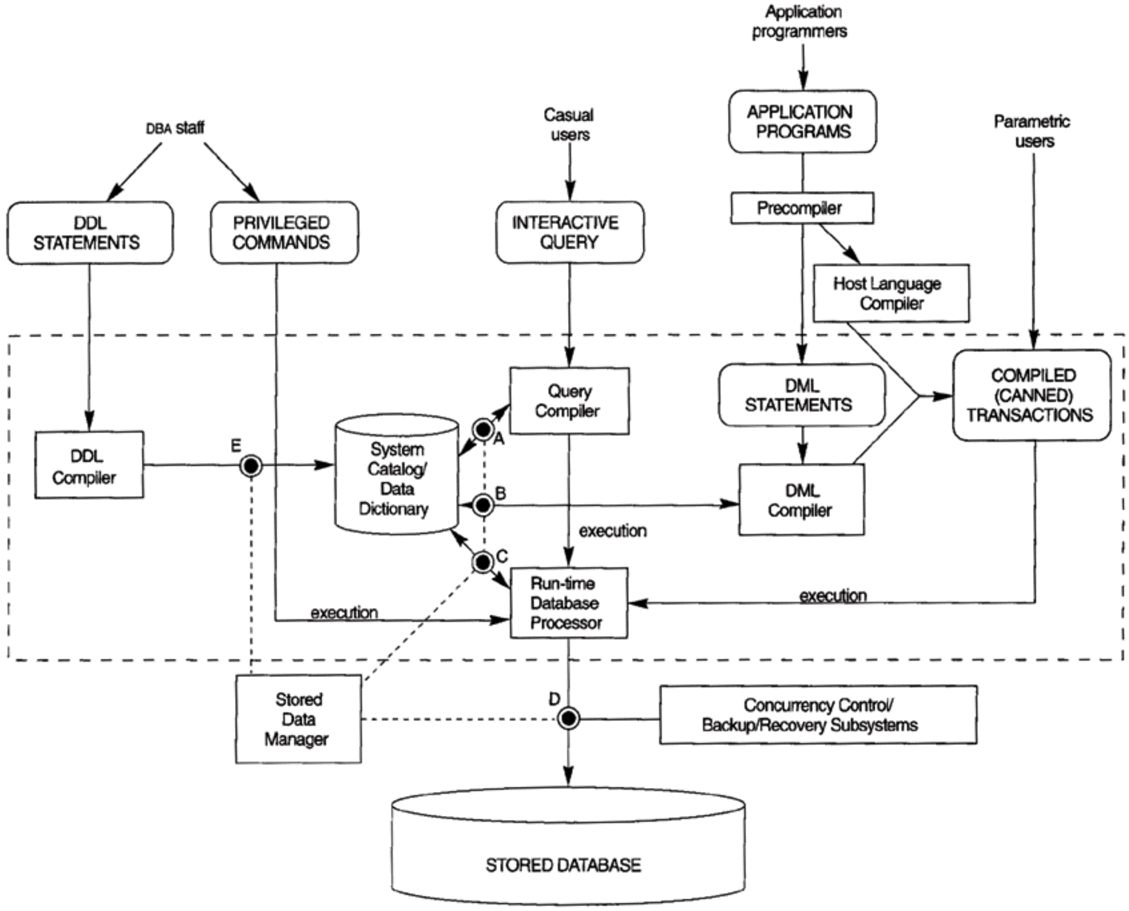


Figure 6: DBMS Languages [Elm 94]

3.2. Relational Database System

A relational database system is a database system implemented based on the relational model [Elm 94]. The relational model uses the concept of a mathematical relation of set theory and first order predicate logic. The database is represented as a collection of relations in the relational model. A relation is usually presented as a table of values. Each row in the table is a collection of related data values. The user can also specify relational constraints and schemas to restrict the data on a relational database. The set of operations for the relational model is known as the relational algebra. Due to its simplicity and mathematical foundations, the relational model has become the most successful model in database development. The relational database systems, which implement relational model, are the main stream of commercial database systems for the last twenty years [Sto 96].

3.2.1. Relational Database Design Process

A useful database must contain accurate, complete and organized data. To achieve these goals, a database should be well designed to avoid inaccurate and inconsistent data. A poorly designed database often has un-necessary duplicated data, which causes data inconsistency and other various problems [Mul 99]. A proper database design contains loops of steps of requirements collection and analysis, conceptual design, logical design, physical design. The physical design is then further refined through usage refinement. The application design process often runs in parallel with the database design and in conjunction of the software engineering process [Elm 94]. In the requirements collection and analysis step, the designer should focus on the domain of the miniworld and requirements of the database. These requirements should be specific and expressed clearly in natural language. In the conceptual design steps, the miniworld and requirements are translated into Entity-Relational (ER) diagrams and relational constrains. The ER diagram was proposed by Peter Chen as a graphical notation display for database schema. [Che76] Based on the conceptual schema generated, the logical schema was formed following the logical design also known as the data model mapping. Finally, the internal storage structures, access paths, and file organizations for the database files are specified in the logical design. Throughout the life of the database, the physical design is modified often to improve performance. On the other hand, the logical design is rarely changed.

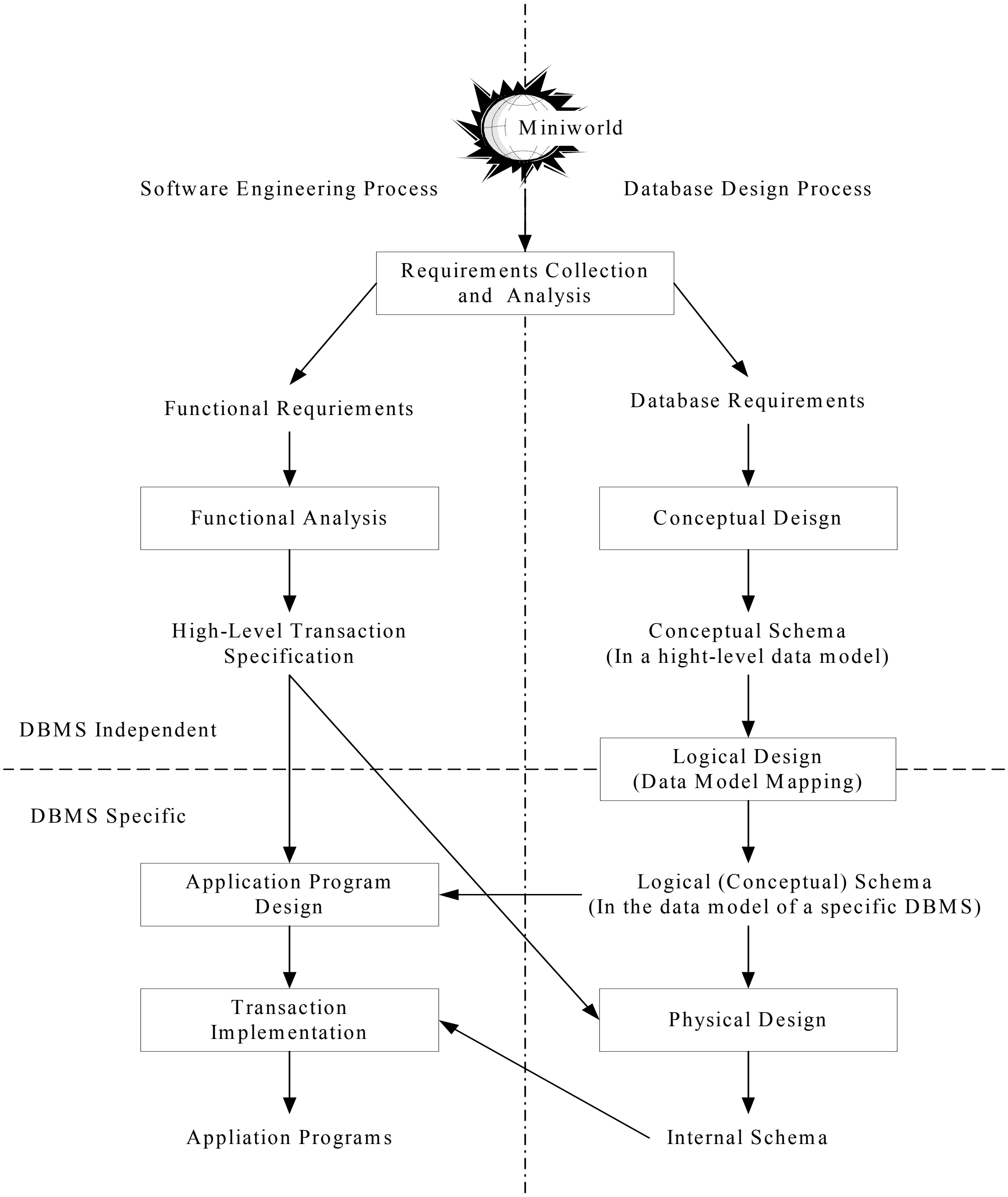

Miniworld
Software Engineering Process
Database Design Process
Requirements Collection and Analysis
Functional Requriements
Database Requirements
Functional Analysis
Conceptual Deisgn
High-Level Transaction Specification
Conceptual Schema (In a hight-level data model)
DBMS Independent
Logical Design (Data Model Mapping)
DBMS Specific
Application Program Design
Logical (Conceptual) Schema (In the data model of a specific DBMS)
Transaction Implementation
Physical Design
Appliation Programs
Internal Schema

Figure 7: DBMS Design Process [Elm 94]

3.2.2. Relational Algebra

A basic set of relational model operations constitutes the relational algebra. These algebra operations are performed on relations, and the result of these operations is a new relation. A sequence of relational algebra operations forms a relational algebra expression. The relational algebra operations are divided into two groups, set operations and relational operations. The set operations are defined from mathematical set theory and include UNION, INTERSECTION, DIFFERENCE, and CARTESIAN PRODUCT [Elm 94]. These operators, except CARTESIAN PRODUCT, are defined on two union-compatible tables. The UNION operation combines two tables by creating a new table including all rows that are in either one or both of the tables. Since the result is a relation, a set, duplicated rows are eliminated. The INTERSECTION operator creates a new table where only the rows that are in both tables exist. The SET DIFFERENCE operation creates a table containing the rows that are in first table but not in the second table. The CARTESIAN PRODUCT operator can be applied to any two tables. The result is all possible combinations of rows from each table. The CARTESIAN PRODUCT is also known as the CROSS PRODUCT or CROSS JOIN. The other group, relational operations, consists of operations developed specifically for relational databases. SELECT, PROJECT, and JOIN are the operations in this group. The SELECT operation is used to select a subset of the rows from a table that satisfy a selection condition. The selection condition is logical expression that can only be either true or false when applied to the rows. PROJECT selects a subset of columns from a table based on the selection. The JOIN operator combines related rows from two tables into single rows. Although relational algebra is complete for set operations, not all database requests can be performed with the basic relational algebra operations. For instance, a user cannot request to count the number of rows in a table or sort the rows in order. As a result, additional operations are implemented by DBMSs [O'Ne 01]. Also it is impossible to express recursive operations in relational algebra. These operations must be implemented by the application if they are not supported by the RDMBS directly. More limitations of the relational algebra are discussed in section 3.3.1.

3.2.3. Structured Query Language

Structured query language (SQL) is a comprehensive database language based on the tuple relational calculus [Sund 00]. Being comprehensive means that SQL is both a DDL and a DML. As a DDL, SQL has statements like CREATE TABLE and NOT NULL to create relation and define constrains. SQL, when used as a DML, has statements like UPDATE and SELECT to update and query the database. Furthermore, SQL has the ability to create view, specify security, authorization, and transaction controls, and define meta-data [Sund 00]. It also has rules for embedding SQL statements into programming language like C or PASCAL. Object-oriented languages also support SQL through standard API and vendor specific drivers [Cat 97]. Other than being comprehensive, SQL also has user-friendly syntax. SQL is the standard database language for relational databases and is also used in some object-oriented databases and object-relational databases [Sto 96; For 99]. SQL being the standard language has contributed to the success of relational databases. Because SQL is a standard, applications developed for relational database are more portable. Users also have fewer difficulties to access data stored in multiple relational DBMSs. Furthermore, SQL provides a high-level declarative language interface that allows user to specify only the results of the procedures. The actual procedures to generate the results are decided by the DBMS. This takes the execution and optimization of the query away from the application. As a result, the application implementation is simplified because the query is optimized without further implementation in the application.

3.3. Third Generation Database Technology

3.3.1. Shortcoming of Relational DBMS

Although relational DBMSs, the most successful second-generation databases, have performed successfully in information technology for the last twenty years, information systems today demand more support for complex objects [Sto 96]. As the miniworld is required to be closer and closer to the real world, the database and the data model it is based on must represent the meaning of data in terms of both structure and behavior. The structural meaning of data represents the entities in the miniworld and how the entities are related. For instance, a sale order is related to customers and products. This information can be represented by relating data values corresponding to a particular structure. In other words, any sale order can be related to customers and products using a particular structure. On the other hand, information can also be analyzed in terms of behaviors. For instance, a sale order causes customers to pay and products being produced. The behavior of an entity can be represented as a set of procedures, a program, that operate on data in the entity. Similar to the structural meaning, a particular entity can be represented by a particular program.

The second-generation DBMSs heavily emphasize the structural meaning of data. Data is stored as entities. However, as an information system evolves, databases are required to store data's behavior meaning as well. Currently, in object and rule based applications like computer aided design (CAD), geographical information systems, and multimedia, databases are required to store and manipulate complex data that is not supported in a relational DBMS [Sto 96]. The third generation DBMS starts to address the need to store an entity's behavior as part of the entity's data. The dominant approach is to store data as an object based on the object-oriented methodology [Pap 00]. An object has attributes and methods where attributes represent structure and methods represent behaviors. The entities in the second-generation DBMS can be seen as an object with only attributes in built-in types. Both Object-Oriented data model and Object-Relational model have been the result of the object approach. Both data models incorporate the object-oriented concept but have different approach applying the concept. The evolution of the data models that the different generations of DBMSs are based on is shown in Figure 8.

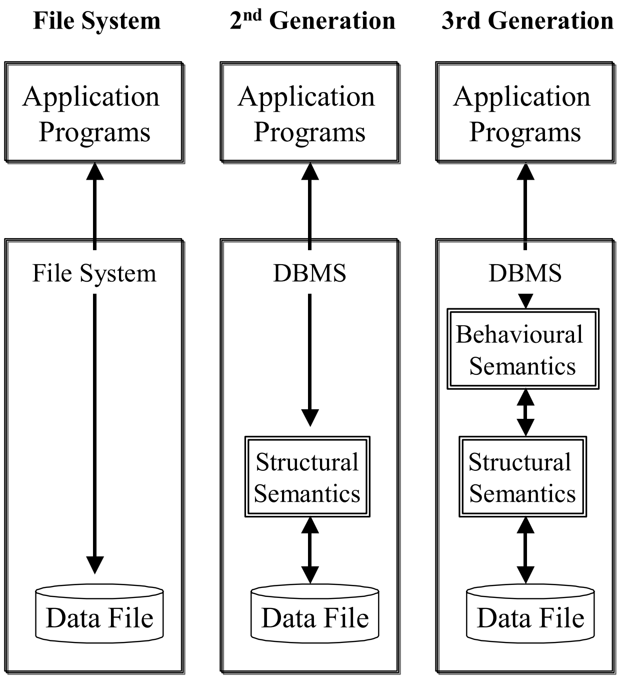


Figure 8: Evolution of Data Models [Pap 00]

3.3.2. Object-Oriented DBMS

Object-Oriented DBMS (OODBMS), also known as Object DBMS (ODBMS), is a database that implements based on the object data model [Pap 00]. The OODBMS addresses second generation database's limitations by allowing users to define objects and methods to manipulate objects in the DBMS. More information, both structural and behavior, is represented by incorporating the facility of object-oriented programming languages [Pap 00]. The users can extend the type system to include new types specially tailored to their applications using object-oriented programming to implement the object's attributes and methods. Comparing with the second-generation databases where behavior information is implemented as part of the application program, an object's behavior is inseparable from its data in an OODBMS [Pap 00].

3.3.3. Object-Relational DBMS

On the other hand, ORDBMS is based on a more evolutionary approach where the relational model is extended to include complex objects [Sto 96]. The extended relational model is often known as the object-relational model. Other than allowing users to define customized objects, ORDBMS also inherits the relational model used in the RDBMS. As a result, the customized objects can also been used in relationships (tables), like the basic built-in types. Since the object-relational model only extends the relational model rather than trying to replace it, many technologies used in the RDBMS can still be applied to ORDBMS [For 99]. Thus, many RDBMS vendors have taken this approach and evolve their RDBMSs into ORDBMSs [Sto 96].

3.4. Choosing A Suitable DBMS Framework

3.4.1. Disadvantage of RDBMS

The relational model has several inherent limitations that make RDBMSs inappropriate for many applications. First, all information in a relational database is represented in relationships of atomic values. This restriction is known as the first normal form where all tables should be normalized [Sto 96]. Due to its simplicity, complex structures existing in the real world are difficult to represent in this tabular form. A complex structured object must be represented by a number of separate tables interconnected by foreign references (i.e., references to rows in other tables). Not only it is difficult to flatten and fragment complex structures correctly and accurately, users also have difficulties to perceive the structure using these fragmented data. Consequently, the application must join all these tables to retrieve the information about one object. The need for recombining information caused the application to be complex. All the nested objects need to repeatedly apply queries also makes the application inefficient.

The other major disadvantage of a relational DBMS is its inability to represent behaviors. Relational database languages only express some aspects of how data can be manipulated as defined in the set theory. Although relational database languages, for example, the domain relational calculus, are relationally complete, the language is not computationally complete [Bla 98]. In other words, relational languages can express everything that can be expressed in the relational algebra in the relational model, but it cannot express arbitrary complex computations. When arbitrary computation is required, the relational database system must rely on programming language such as C. Thus, relational language must be included within programming languages, such as C or Ada. In object-oriented programming languages, relational database languages are passed to RDBMS using connectivity API, like JDBC or ODBC, and vendor specific drivers [O'Ne 01]. In either case, these procedures are part of the application instead of the relational database since the database only supports structural data. Behavior data, usually expressed as arbitrary computational procedures, must be stored within the application. The application can become very complex in order to express and manage these behaviors.

3.4.2. Disadvantage of ODBMS & ORDBMS

The third generation databases, both ODBMS and ORDBMS, have addressed the disadvantages of the RDBMS. However, the technology is relatively new compared with RDBMS, and this could be the major disadvantage of ODBMS and ORDBMS. The object data model, on which ODBMS is based , exists in a confusing variety of forms, with different and sometimes contradictory terminologies and definitions [Cat 97]. On the other hand, ORDBMS extends the relational model and adds a significant amount of complexity into the model [For 99]. Furthermore, both ODBMS and ORDBMS lack a standard database language, like SQL in RDBMS. Currently, the Object Specification Languages (OSL) and Object Query Language (OQL) specified by the Object Data Management Group (ODMG) standard dominate the ODBMS market [Cat 97]. The SQL3 dominates the ORDBMS area [For 99]. However, vendors have had little time to implement their DBMS to be standard compliant. Thus, most commercial ODBMS and ORDBMS are still vendor-specific. This disadvantage has made applications using third generation DBMS much less portable than application using RDBMS.

3.4.3. A Suitable Database Framework

Although it is difficult to store complex structures and behaviors in a relational database, complex structures can be decomposed and re-constructed in the application and minimum behavior storage is required in SESM. Detailed description of how these limitations affected the SESM and how SESM overcomes these limitations is presented in Sections 5.6 and 5.7. Furthermore, the relational model is mathematically rigorous which allows formal specification of the logical schema of SES. The simplicity of the relational model also allows the DBMS vendors to optimize the management systems for performance and scalability. Compared with relational database, the third generation databases, OODB and ORDB, are still constantly evolving. If these databases were chosen as the repository, the SESM would have to be constantly updated as its database evolves. This would add unnecessary uncertainty and maintenance work to the application development. Furthermore, both ODBMS and ORDBMS lack the full support for the current standards. The lack of standardizations means less support for developing SESM, and it makes the developed SESM vendor-specific. As a result, after comparing the three types of database technologies, the relational database was selected to be the storage medium for SESM.

4. SESM ARCHITECTURE SYSTEM VIEW

4.1. A Model Representation and Management Methodology

As mentioned in Chapter 1, Scaleable System Entity Structure Modeler should support multiple users concurrently accessing and manipulating models that are stored in the database. Thus, the SESM must provide the management mechanism to keep the data consistence. Based on the SESM specification, several types of distribution architecture were considered. The three architectures that were compared are client-server, advanced client-server, and hybrid architecture.

4.2. Client-Server Architecture

The client and server architecture uses the SESM Server as the manager. Clients must submit their modification to the server through the network environment (see Figure 9). The main functions of the network environment are initializing connectivity between clients and sever and serializing the messages passed between them (see Figure 10). When the server receives a request, it modifies the database accordingly using the DBMS. If the modification was a success, the server notifies each client about the change. This architecture ensures that only the server modifies the database. Depending on the capacity of the network, one of two methods of updating could be implemented. In the first method, the server only broadcasts the change that was made to the model data. When clients receive the changes, they update their local copy of the model data accordingly. The other method is that the server broadcasts a notification to the clients when a modification has been made. Whenever the clients received the notification, each retrieves a copy of the model data from the server. Both methods are shown in Figure 10. The advantage of the first method is less dependency on the network capacity. Since only the changes are broadcast, the size of each message is much less compared with the other method. However, when the client initializes, it still needs to retrieve its local copy from the server. The main drawback is that the client must maintain its own local copy. This will greatly increases the complexity of the client. Overall, the client and server architecture does not utilize the DBMS's efficiency and scalability. The server provides similar functionality as the DBMS. Yet, it is difficult for the server to provide the same performance as the DBMS. Plus, in order to communicate with each other, the client and the server must agree upon a fixed data structure for storing the model data. The fixed data structure eliminates the possibility to implement the client and the server independently.

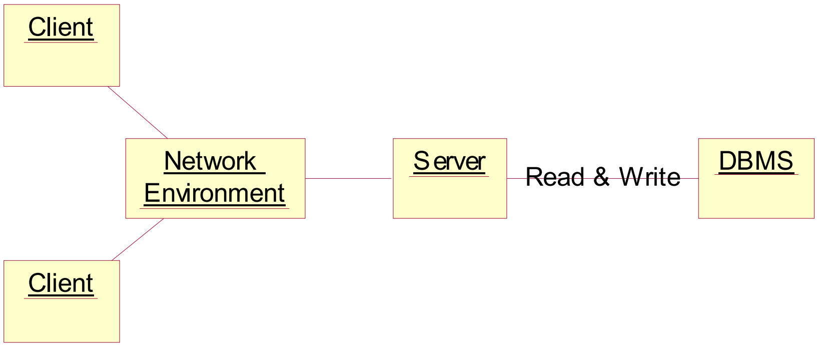


Figure 9: Client/Server Architecture

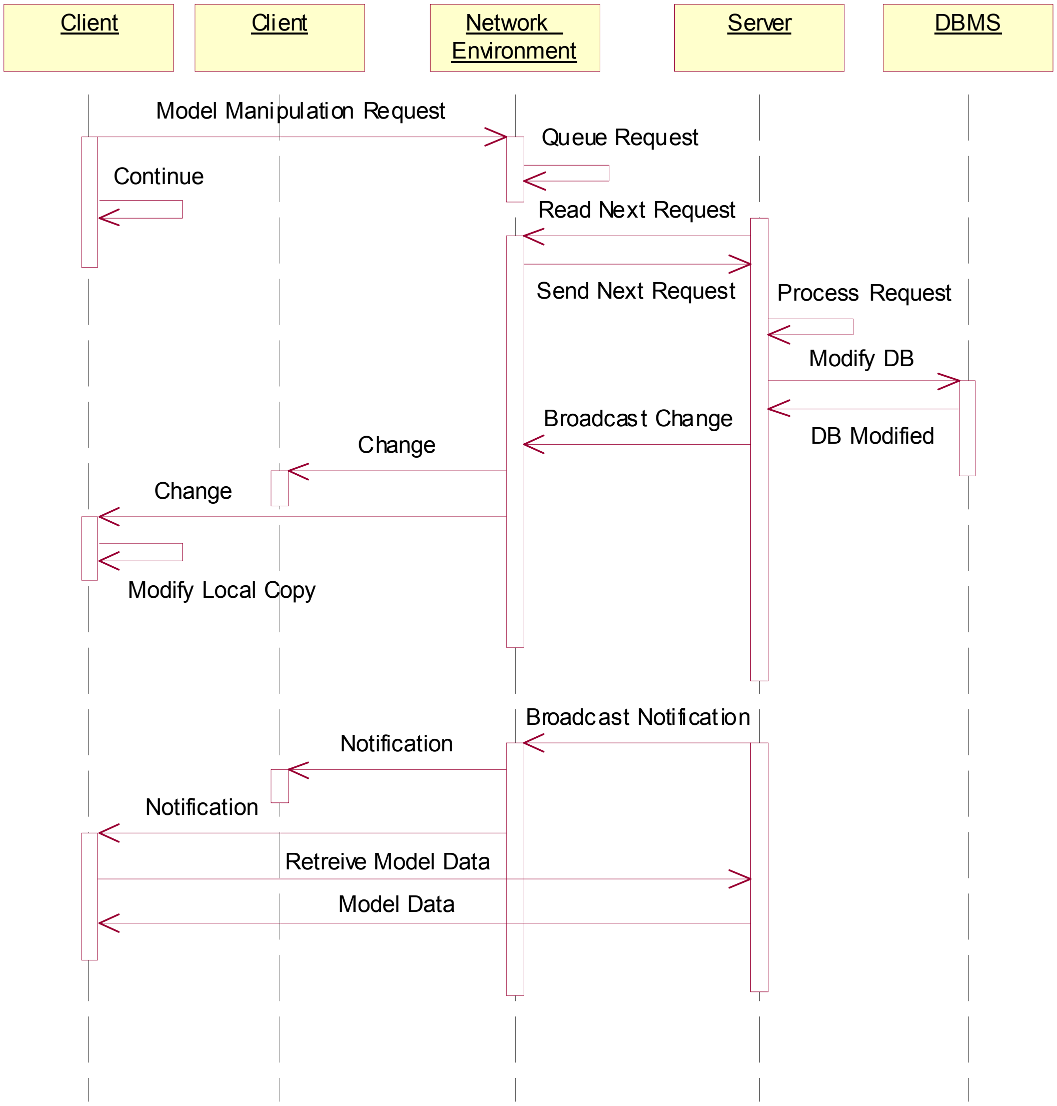


Figure 10: Client/Server Architecture Interaction

4.3. Advanced Client-Server Architecture

This architecture allows each client to connect to the DBMS directly. In this architecture, all clients are allowed to read from and write to the database where each client must ensure atomicity of each transaction. This requires the client application to implement and use some appropriate locking mechanism for any modification to the database. This architecture is generally more efficient than the simple client/server architecture since it allows multiple Readers and multiple Writers access the database concurrently. Most commercial DBMS support some form of mediated multiple clients and server interactions. However, this architecture requires additional complexity for the client since each client must ensure the atomicity of its transaction w.r.t. other collaborating clients. Due to proprietary/customized locking protocols and implementations (collaboration approach) by each database management system, clients must be specified and implemented for each commercial DBMS.

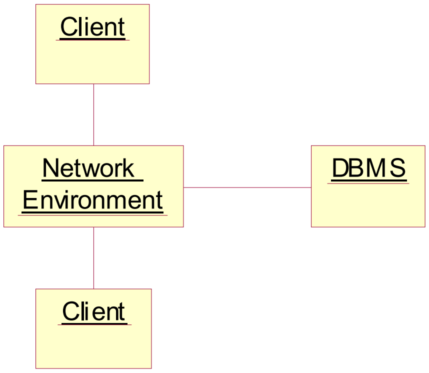


Figure 11: Advanced Client/Server Architecture

4.4. SESM System Architecture

4.4.1. Hybrid SESM Architecture

Both the client/server architectures described above have their advantages and disadvantages. Thus, the SESM uses a hybrid of the two above architecture to reach a balanced solution. The SESM architecture shown in Figure 1 is repeated as Figure 12 below. Similar to the client/server architecture, the hybrid architecture has a server that writes to the database, with potentially multiple clients interacting with the database. On the other hand, similar to the advanced client/server architecture, clients are also connected to the DBMS, allowing clients to retrieve model data concurrently. Yet, the client cannot write to the database. The client must connect to the server in order to modify the database. This architecture allows a single writer but multiple readers to be in the database concurrently and thus provides a restricted flavor of network environment.

4.4.2. Component Interactions

The client can modify the model data by sending a request with required parameters to the server (see Figure 15). In this architecture, the clients' requests to the server are serialized if they are intended to modify the model. The serialization ensures that the server only receives a request at any time. When the server receives a modification request, the server broadcasts a notification to all the clients. When each client receives the server notification, the client reads the model data directly from the DBMS. However, if the server did not complete the modification, the server sends out a message to the specific client that requesting the modification with a reason of failure. Thus, the client also receives feedback of its incomplete tasks from the server.

4.4.3. Advantages of SESM Architecture

Comparing with the client/server architecture, the hybrid architecture produces less network traffic between the clients and the server. Thus, it provides better scalability since a large number of queries have been shifted from the server to the DBMS. Retrieving the model data directly from the database has an additional benefit. It makes the development of the client and the server less dependent. Since the client will not receive model data from the server, a uniform data structure to store the model data is not needed. The model data representation by the client and the server can be independent from each other except for the simple messages that they exchange. That is, each client can be developed to store their model data differently from the server and each other. Furthermore, unlike the advanced client/server architecture where each client acts as both reader and writer, the hybrid architecture specifies distinct functionalities for the client and the server. This allows modular design and implementation and thus helps the development process.

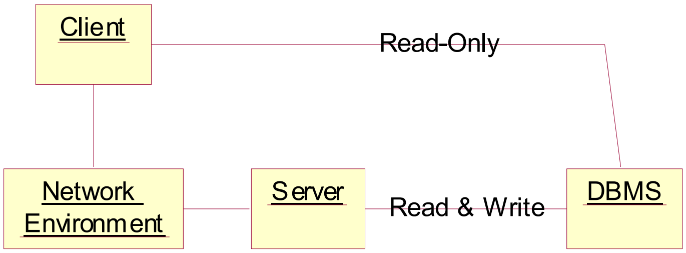


Figure 12: Scaleable System Entity Structure Modeler

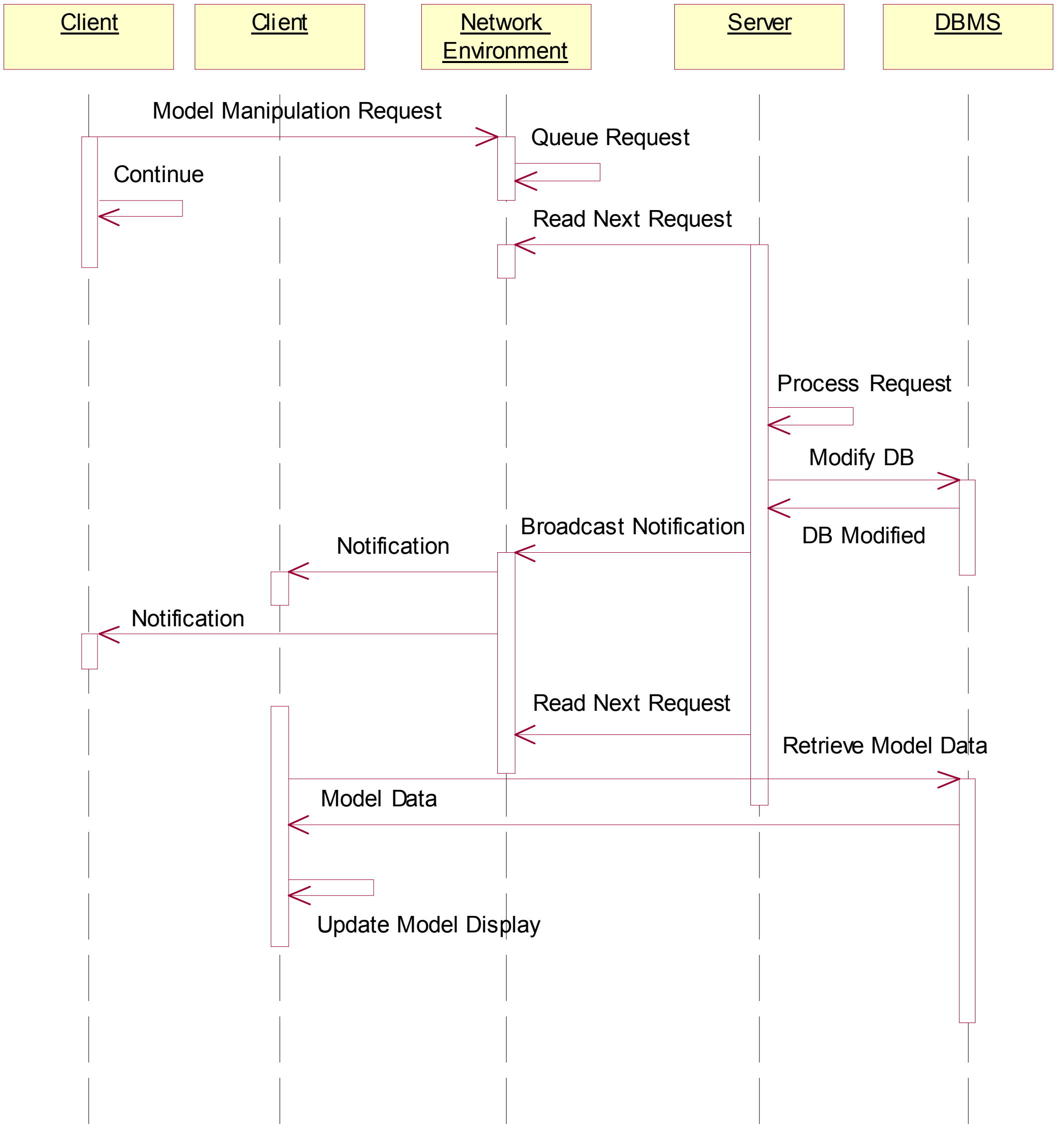


Figure 13: SESM Architecture Interaction Diagram

4.5. SESM System Design Overview

By design, the SESM system includes the SESM package, the Network Environment package, SESM client, and SESM server as shown in the SESM System Components Overview Diagram. The SESM package should serve as an API used to access the SESM representation model data stored on the DBMS. There are three main components in the SESM package; Connectivity, SESM Query, and SESM Modifier.

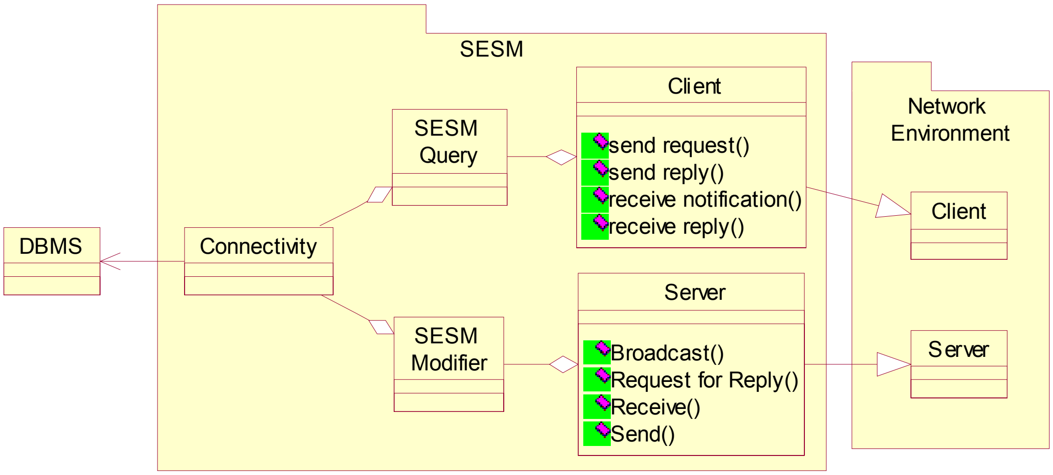


Figure 14: SESM System Components Overview Diagram

The connectivity component is used to connect to the DBMS. It handles all the communication between the SESM system and the DBMS. The SESM Query component retrieves data from the DBMS using SQL and maps the data into object-oriented SESM models. The SESM Modifier component modifies the SESM representation models on the DBMS. The component translates the requested modification into appropriate SQL statements. The SESM Server extends the Server provided by the Network environment package. Messages received by the SESM Server are processed, and modifications are performed accordingly. The SESM Client utilizes the SESM Query component to retrieve and display the SESM representation model visually on its graphical user interface (GUI). The user also modifies the model through the SESM client's GUI. The network Environment package manages the communication between the SESM Client and the SESM Server by providing the components that can be extended by the SESM Client and SESM Server. The Client component should have "non-blocking send" and "blocking receive" methods allowing the SESM Client to send message to, and receive message from, the SESM Server. The Server component should have "non-blocking broadcast", "blocking receive", and "request" methods. The request method sends a message to a specific client and waits for the client to reply. For instance, a client requests a transaction to create an Instance Model from a Specialized Model Template. The server needs to ask the client to specify a Specialization Template Model by providing a list of Template Models. The transaction cannot be continued until the client has specified the Specialization Template Model. Also, the client who specified the Specialization Template Model should be the same client that requests the transaction. As a result, the request interaction is a one-to-one interaction between the server and a particular client. The receive method allows the server to receive one message at a time from any client. The server uses the broadcast method to broadcast a message to all clients that are connected to the server. The broadcast method should be a non-blocking send where the server does not wait for any reply from the client.

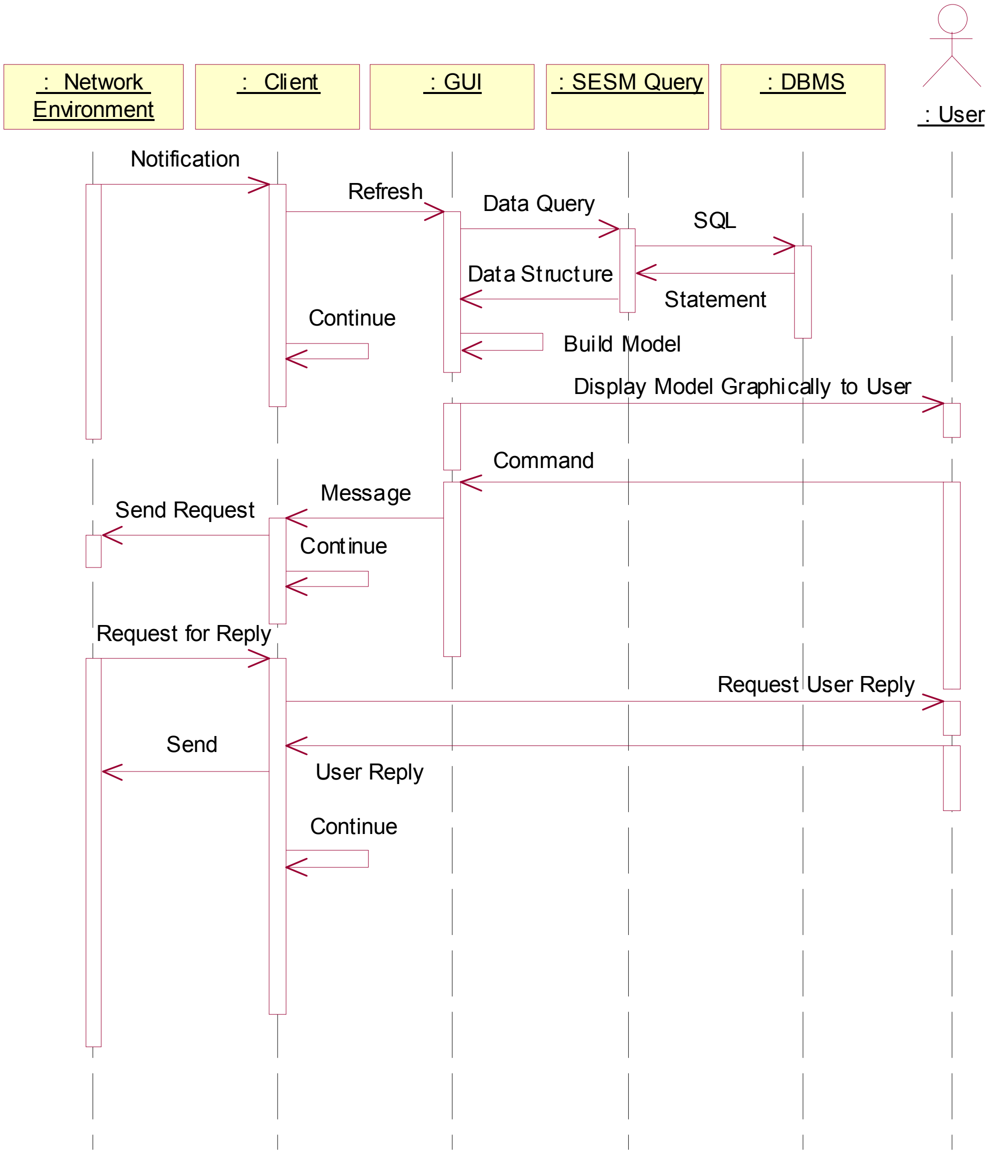


Figure 15: SESM System Client Interaction Diagram

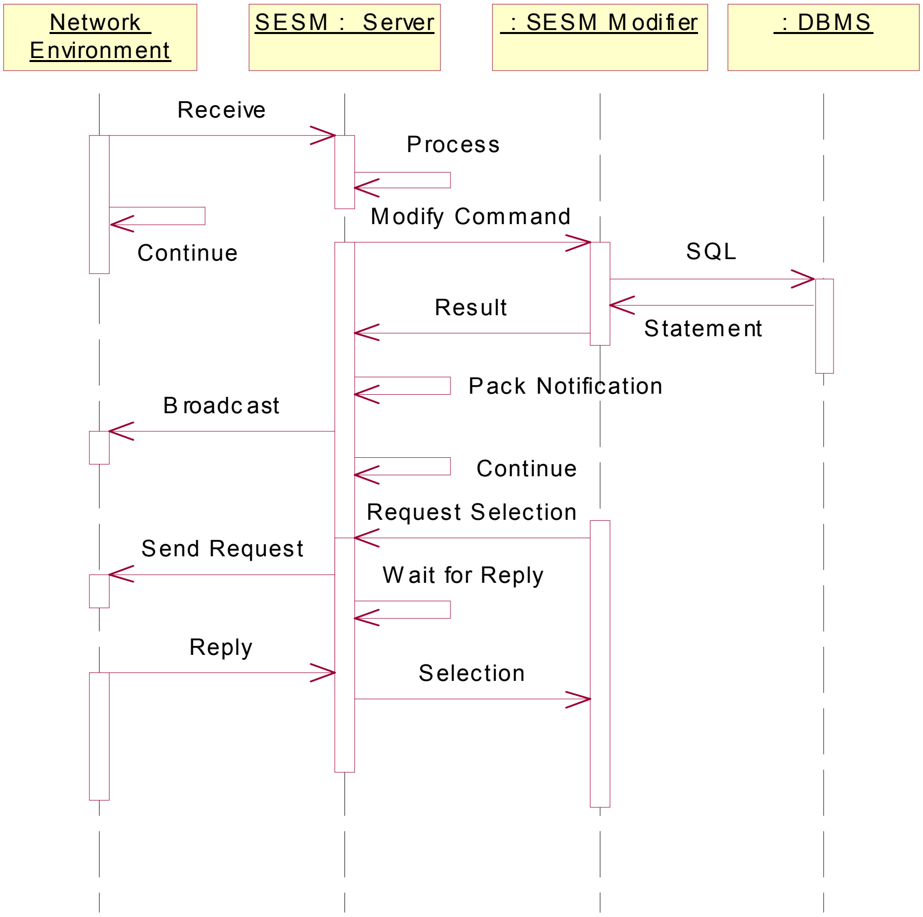


Figure 16: SESM System Server Interaction Diagram

5. MODELING APPROACH

In this chapter, we present a new approach to specifying hierarchical models in the spirit of the SES approach. We present SESM requirements and its relational database schema based on the modeling approach. The details of the SESM entity relationship diagram (e.g., specification of template model and its relationships with other models) are presented and exemplified using the Switch Network example introduced earlier.

5.1. Representation Approach

To enable specifying a family of models in an incremental and systematic approach, we can use three classes of model: Template Model, Instance Template Model, and Instance Model [Sar 02]. With these classes of models, we can represent a family of hierarchical models with input/output interface, ports, and couplings in a systematic fashion.

Template Model (TM)**:** A template model can be either an atomic or a coupled model each with its unique name. Each coupled model can have as its elements a finite number of other template models provided that the **depth of each template model is either one or two**. The allowed relationships among template models are "has-part" and "kind-of". Every template model is unique structurally w.r.t. all other template models in a given family of models. Every template model structure is defined in terms of its input/output interface (ports), the elements it may have via part-of relationship, and the coupling that may exist among all elements (children and parent/child). The has-part relationship only allows parent to child relationship (e.g., grandparent/child relationship is not allowed). Relationships among children are specified using couplings. Parent/child relationship (has-part relationship) is also specified via couplings. Couplings are required for any two models to have valid well-defined relationship with one another. Kind-of relationship denotes specializations of a template model in terms of other template models.

Instance Template Model (ITM): An instance template model is an instantiation of a set of template models. An instance template model can be an instance of a template model or a synthesis of two or more unique template models. An ITM represent models with arbitrary, but finite, depth. All instances of a template are distinguishable from one another. Each instance template model has a unique ID in addition to what it inherits from TM (structure and unique name). ID is necessary to distinguish two instances of a template model having the same name. Instance template models can have has-part and kind-of relationships. Since ITM supports **representing multi-level hierarchical models**, child and grandparent relationship can exist. The couplings between ITM components are the same as those specified in the template models. No new couplings can be introduced in an instance template model – such couplings must be specified in the template models.

An **Instance Model** (IM): an instance model is **an instantiation of a template instance model**. Each instance model is an exact copy of a template model. All instances of an instance template model are distinguishable from one another. Each instance model has its own ID that can be used to distinguish it from other instance models with the same name.

The process of specifying/constructing model structures using template, instance template, and instance models supports an alternative approach in developing models based on the System Entity Structure. This approach allows the user to incrementally develop models – incorporate multi-layer hierarchical models using Instance Template Model and allowing multiple instances using Instance Model.

5.2. SESM Requirements

The following accounts for a set of requirements for model development based on the above three categories of model structures and the user-specification of CDM [Sar 99]. These requirements help in the specification of models and their implementation in a relational database.

Model

- A model can be a Template, an Instance Template, or an Instance
- Any of Template, Instance Template, or Instance model can be either atomic or coupled; an atomic model cannot be decomposed; a coupled model may be decomposed into one or more other (atomic or coupled) models
- A model cannot contain itself as a component

- A model may have one to N ports
- Models may be specialized; a Specialized Model can be specialized into 1 to N Specialization Models
- A Specialization Model can only specialize one Specialized Model
- When a Specialized Model is deleted, its Specialization Models will not be deleted
- When a Specialization Model is deleted, its specialized Models will not be deleted
- A model may either be specialized or decomposed but not both
- Models may be sorted by their creation time

Port

- A port must have a type (Input or Output) and a name
- A port belonging to a model must be uniquely identifiable by its type and name
- A port may be coupled to 0 to N other ports of one or more other models
- A port cannot exist without being assigned to a model
- Ports may be sorted by their creation time

Coupling

- A coupling must link two ports of different models (self coupling is not allowed)
- A coupling may exist between components of a coupled model or between the coupled model and any of its immediate components (parent/child coupling)
- A coupling should couple same type of ports (input to input and output to output) when the coupling is between a coupled model and it component
- A coupling should couple input to output or output to input when the coupling is between two component models of the coupled model

Template Model

- A Template Model must be uniquely identifiable by its alphanumerical name; the name of a Template Model is assigned by its creator
- A Template Model must exist before one or more of its Instance Models can be created
- When a Template Model is deleted, all its Instance Template Models and Instance Models must be deleted

- The depth of a Template Model hierarchy can be most two – no grandchild is allowed

Instance Template Model

- An Instance Template Model must have a name (alphanumerical) and a unique identifier/ID (integer)
- An Instance Template Model can only be created from one Template Model
- Multiple Instance Template Models created from the same Template Model should be uniquely identifiable by their IDs
- An Instance Template Model must have a tree structure
- An Instance Template Model may be a component of at most one other Instance Template Model with itself appearing only once in any branch of its tree structure
- An Instance Template Model can be used to create zero to N Instance Models

Instance Model

- An Instance Model must have a name (alphanumerical) and a unique identifier/ID (integer)
- An Instance Model can be created from one Instance Template Model
- Instance Models created from the same template must be uniquely identifiable by their IDs
- An Instance Model structure (components, ports & couplings) must be identical to the Instance Template Model from which it is instantiated
- An Instance Model may contain zero to N Instance Models as its components with itself appearing only once in any branch of its tree structure
- An Instance Model may be contained in multiple coupled Instance Models
- Multiple Instance Models with the identical structure may be contained in a coupled Instance Model
- When an Instance Model is deleted, all its components are deleted; only an Instance Model that is not a component of another Instance Model can be deleted
- An Instance Model cannot be a Specialized Model. A Specialized Model must be specialized by one of its Specialization Models for it to be an Instance Model.

5.3. SESM Entity-Relational Diagram

Based on the above requirements, the ER diagram shown below was developed.

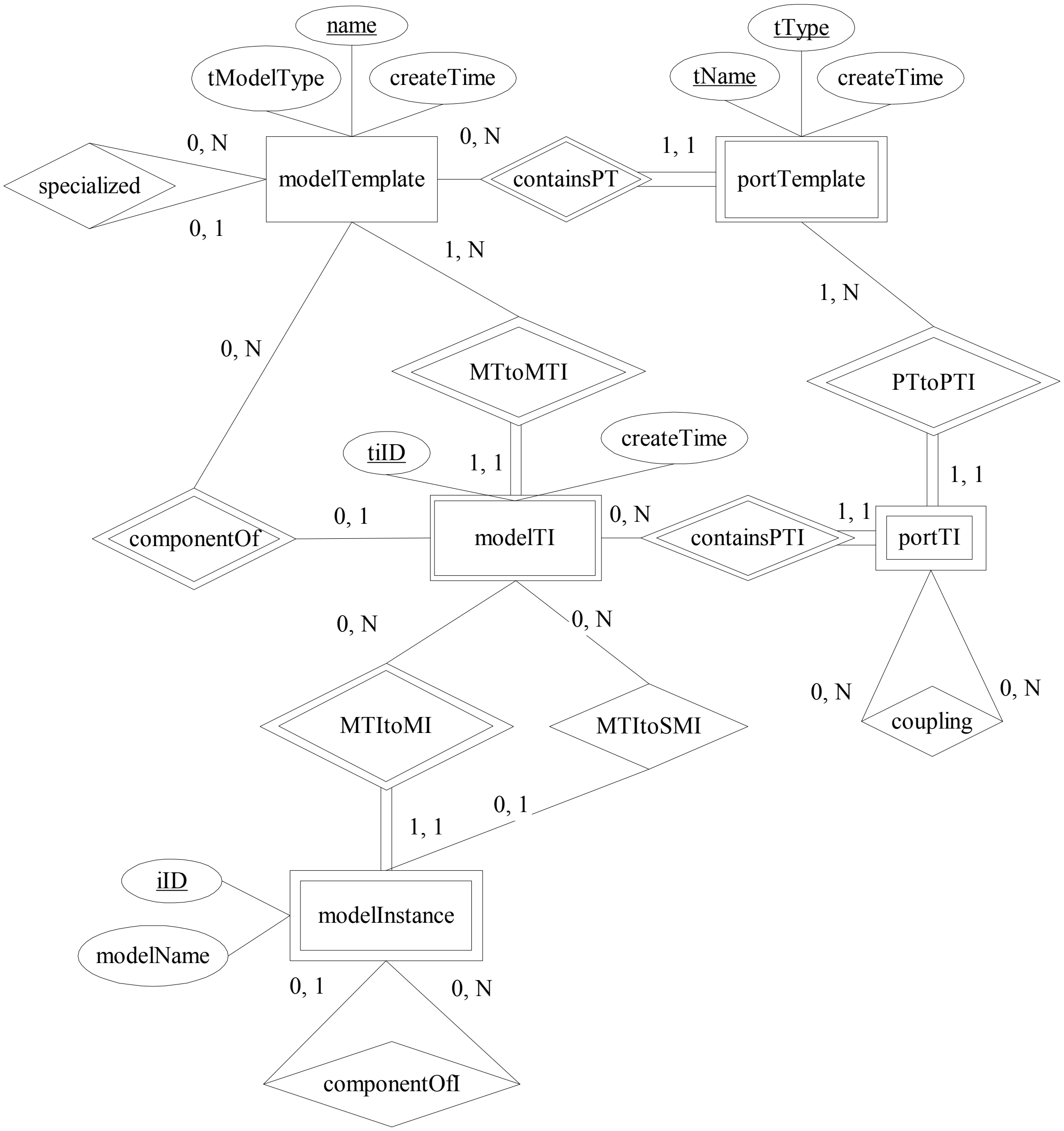


Figure 17: SESM ER Diagram

Label	Note
containsPT	Template Model contains Template Port
containsPTI	Instance Template Model contains Instance Template Port
iID	Instance Model ID
modelTemplate	Template Model
modelTI	Instance Template Model

MT to MTI	Template Model to Instance Template Model
MTItoMI	Instance Template Model to Instance Model
MTItoSMI	Instance Template Model to Specialization Instance Model
PortTI	Instance Template Port
PTtoPTI	Template Port to Instance Template Port
tiID	Instance Template Model ID

5.3.1. Entities in SESM Entity-Relational Diagram

modelTemplate (Template Model)

- Attributes
  - name (Template Model name)
  - tModelType (Template Model type)
  - createTime (time of Template Model creation)
- Descriptions:
  - The modelTemplate entity represents the Template Model in the SESM specification
  - All attributes are single-valued and not null-able
  - The attribute, name, is the primary key for modelTemplate since Template Model is uniquely identified by its name. The value set of this attribute is alphanumerical string.
  - The attribute tModelType is the type of the Template Model. The values set of this attribute is string of 'ATOMIC', 'COUPLED' and 'SPECIALIZED'.
  - The attribute createTime records the time when the Template Model was created. It can be used to sort Template Models.

portTemplate (port template)

- Attributes
  - tName (port name)
  - tType (port type)
  - createTime (time of port creation)
- Descriptions:
  - The portTemplate entity represents the ports belong to the Template Models.
  - The portTemplate is a weak entity of modelTemplate because a portTemplate should not exist if the modelTemplate contains it does not.

- The modelTemplate and portTemplate is linked by containsPT (contains port template) relationship.
- All attributes are single-valued and not null-able
- The attribute, tName, is the name of the port and a partial key of the portTemplate. Its value set is alphanumerical string.
- The attribute tType is the type of the port and a partial key of the portTemplate. Its value set is string of 'IN' and 'OUT'. This restriction is defined by DDL when the table is initialized in the database.
- The attribute createTime records the time when the port template was created. It was used to sort ports in SESM.

modelTI (Instance Template Model)

- Attributes
    - tiID (instance template identification)
    - createTime (time of Instance Template Model creation)
- Descriptions
    - The Instance Template Model is defined to represent a two-level model structure. With multiple appearances of the same Template Model, each appearance must be uniquely identified in order to support coupling specifications. An Instance Template Model (modelTI) plays an important role in capturing coupling information.
    - modelTI is a weak entity because it is created from modelTemplate.
    - The attribute tiID is a partial key of the modelTI. Its value set is integer. The integer value should be assigned by the application.
    - The attribute createTime records the time when the Instance Template Model was created.

portTI (port template instance)

- Attributes: None
- Descriptions
    - The entity portTI is defined alone with the modelTI. It represents the ports contained by the modelTI.

  - portTI has no attributes since its foreign attributes are sufficient to uniquely identify it.
  - portTI is a weak entity because it is created from portTemplate and created for the modelTI. As specified in the requirements, a port should not exist without its model. Plus, all Instance Template Models representing the same Template Model should have an identical structure. Thus, portTI is constrained by two identifying relationship, PTtoPTI and containsPTI.

modelInstance (Instance Model)

- Attributes:
  - iID (Instance Model identification)
  - modelName (Instance Model name)
  - createTime (time of Instance Model creation)
- Descriptions
  - The entity modelInstance represents an Instance Model.
  - The attribute, iID, is used to unique identify all instances created from the same instance template. It is a partial key of the modelInstance. Its value set is integer. The integer value should be assigned by the application.
  - The attribute modelName is the name of the Instance Model. Its value set is alphanumerical string.
  - The attribute createTime records the time when the Instance Model was created.

5.3.2. Relationships in SESM Entity-Relational Diagram

specialized

The specialized relationship allows specializing a specialized Template Model to one or more specialization Template Models. The cardinality of this relationship is 0, N and 0, 1 – i.e., zero or more template models can be specialized template models and one or more specialization template models may be defined for a specialized template model. The relationship is partial participation from modelTemplate since Template Models can be either coupled or atomic.

containsPT

containsPT defines the relationship between a Template Model and its ports. It is the identifying relationship of the weak entity, portTemplate. The relationship is a 0, N portTemplate to 1 modelTemplate. Since portTemplate is the weak entity, it has total participation in the containsPT relationship. On the other hand, modelTemplate has partial participation in the relationship because some Template Models might have 0 port.

MTtoMTI

MTtoMTI defines the creation of an Instance Template Model from a Template Model. It is the identifying relationship of the weak entity, modelTI. Both modelTemplate and modelTI have total participation in the MTtoMTI relationship. The modelTI has total participation because it is a weak entity. modelTemplate is required to have total participation so all Instance Models can be created from Instance Template Models.

PTtoPTI

PTtoPTI defines the relationship between Template Model's port structure and its Instance Template Models' port structure. The cardinality of this relationship is exactly one portTemplate to one to N portTI. As defined earlier, Instance Template Model must have the same exact structure as its Template Model. As a result, the ports of an Instance Template Model are created by duplicating its Template Model ports. We note that this relationship does not match the requirements given in Section 5.2 since the PTtoPTI relationship dose not specify Instance Template Model ports to be duplicated.

componentOf

The componentOf relationship represents decomposition. It is the relationship between a coupled model and its components. This relationship is realized by associating an Instance Template Model with its coupled Template Model. The cardinality of this relationship is a 0, N modelTI to 0, 1 modelTemplate. This definition allows the support for multiple appearances of the same Template Model as components of an Instance Model.

containsPTI

containsPTI is the relationship between Instance Template Model (modelTI) and its ports (portTI). The cardinality of this relationship is one modelTI to zero to N portTI. portTI has total participation in the containsPTI relationship since the relationship is portTI's identifying relationship. ModelTI has partial participation for the same reason that modelTemplate has partial participation in the containsPT relationship.

coupling

The coupling relationship represents a coupling between two ports. There are exactly two ports participate in this relationship. Each port can participate in 0 to N coupling relationship as defined in the database requirements.

MTItoMI

The MTItoMI relationship represents the transformation from an Instance Template Model to an Instance Model. It is the identifying relationship of the weak entity, modelInstance. The relationship is a 1 modelTI to 0, N modelInstance relationship. modelTI only has partial participation of the relationship because an Instance Template Models is not required to be transformed into its corresponding Instance Model.

MTItoSMI

The MTItoSME relationship represents the selection from specialized Template Model to a specialized Template Model and its instance. The relationship is a 0, N modelTI to 0, N modelInstance relationship.

componentOfI

The componentOfI relationship represents the decomposition of coupled Instance Models. This relationship allows having multiple instances of the same Instance Model – the instances are structurally identical where each can be uniquely identified by its name in addition to its ID. The componentOfI relationship is a 0, 1 modelInstance (coupled model) to 0, N modelInstance (components). There is no total participation from either sides because an Instance Model can be an atomic model which has 0 component or be at the root level which is not any Instance Model's component.

5.4. SESM Relational Database Schema

5.4.1. SESM Relational Database Schema Specification

Based on the ER-Diagram shown in Figure 17, the schema of the SES relational database was specified as follows. Foreign Keys are shown as ***bold-italic*** and Primary Keys are shown as **bold**. All other column names are shown in plain font.

ModelTemplate		
name	tModelType	createTime

name is an alphanumerical String with maximum length of one hundred characters

tModelType is string and can only be either ‘ATOMIC’, ‘COUPLED’ or ‘SPECILIZATION’

createTime is integer

The primary key is name

PortTemplate			
owner	**tName**	**tType**	createTime

owner is a foreign key from modelTemplate (name)

tName is an alphanumerical String with maximum length of one hundred characters

tType is a string and can be either ‘IN’ or ‘OUT’

createTime is an integer

The primary key is owner, tName and tType

ModelTI		
tID	**tiID**	createTime

tID is a foreign key from modelTemplate (name)

tiID is an assigned integer from 0 to N. tiID 0 is always assigned to the Instance Template Model representing the Template Model. TiID numbers from 1 to N are assigned to the Instance Template Models created as components.

createTime is an integer

The primary key is tID and tiID

PortTI			
tOwner	***tiOwner***	***tName***	***tType***

tOwner is a foreign key from modelTemplate (name)

tiOwner is a foreign key from modelTI (tiID)

tName is a foreign key from portTemplate (tName)

tType a foreign key from portTemplate (tType)

The primary key is tOwner, tiOwner, tName and tType

modelInstance			
template	***templateI***	**iID**	modelName

template is a foreign key from modelTemplate (name)

templateI is a foreign key from modelTI (tiID)

iID is integer assigned by the application from 1 to N for each instance created from a template

modelName is an alphanumerical String with maximum length of one hundred characters

createTime is an integer

The primary key is template, templateI and iID

specialization	
template	***specialization***

template is a foreign key from modelTemplate (name)

specialization is a foreign key from modelTemplate (name)

The primary key is template and specialization

componentOf		
tOwner	***tComponent***	***tiComponent***

tOwner is a foreign key from modelTemplate (name)

tComponent is a foreign key from modelTI (tID)

tiComponent is a foreign key from modelTI (tiID)

The primary key is tOwner, tComponent and tiComponent

coupling							
tOwnerF	***tiOwnerF***	***tNameF***	***tTypeF***	***tOwnerT***	***tiOwnerT***	***tNameT***	***tTypeT***

tOwnerF is a foreign key from modelTemplate (name)

tiOwnerF is a foreign key from modelTI (tiID)

tNameF is a a foreign key from portTemplate (tName)

tTypeF a foreign key from portTemplate (tType)

tiOwnerT is a foreign key from modelTemplate (name)

tiOwnerT is a foreign key from modelTI (tiID) tNameT is a a foreign key from portTemplate (tName)

tTypeT a foreign key from portTemplate (tType)

The primary key is tOwnerF, tiOwnerF, tNameF, tTypeF, tOwnerT, tiOwnerT, tNameT, and tTypeT

MTItoSMI				
tTemplate	***tiTemplate***	***tSpecialization***	***tiSpecialization***	***iSpecialization***

tTemplate is a foreign key from modelTI (tID)

tiTemplate is a foreign key from modelTI (tiID)

tSpecialization is a foreign key from modelInstance (template)

tiSpecialization is a foreign key from modelInstance (templateI)

iSpecialization is a foreign key from modelInstance (iID)

componentOfI					
tOwner	***tiOwner***	***iOwner***	***tComponent***	***tiComponent***	***iComponent***

tOwner is a foreign key from modelInstance (template)

tiOwner is a foreign key from modelInstance (templateI)

iOwner is a foreign key from modelInstance (iID)

tComponent is a foreign key from modelInstance (template)

tiComponent is a foreign key from modelInstance (templateI)

iComponent is a foreign key from modelInstance (iID)

5.4.2. SESM Relational Database Schema in DDL

The SQL statement defining each table, the DDL, is given below.

ModelTemplate

```
CREATE TABLE MODELTEMPLATE (
NAME VARCHAR (100),
TMODELTYPE VARCHAR (13) CHECK (TMODELTYPE IN (‘ATOMIC’, ‘COUPLED’, ‘SPECIALIZED’)),
CREATETIME INTEGER,
PRIMARY KEY (TID)
)
```

PORTTEMPLATE

```
CREATE TABLE PORTTEMPLATE (
OWNER VARCHAR (100),
TNAME VARCHAR (100),
TTYPE VARCHAR (5) CHECK (TTYPE IN (‘IN’, ‘OUT’)),
CREATETIME INTEGER,
PRIMARY KEY (OWNER, TNAME, TTYPE),
FOREIGN KEY (OWNER) REFERENCES MODELTEMPLATE (TID) ON DELETE CASCADE
)
```

SPECIALIZATION

```
CREATE TABLE SPECIALIZATION (
TEMPLATE VARCHAR (100),
SPECIALIZATION VARCHAR (100),
PRIMARY KEY (TEMPLATE, SPECIALIZATION),
FOREIGN KEY (TEMPLATE) REFERENCES MODELTEMPLATE (NAME) ON DELETE CASCADE,
FOREIGN KEY (SPECIALIZATION) REFERENCES MODELTEMPLATE (NAME) ON DELETE CASCADE
)
```

MODELTI

```
CREATE TABLE MODELTI (
TID VARCHAR (100),
TIID INTEGER,
CREATETIME INTEGER,
PRIMARY KEY (TID, TIID),
FOREIGN KEY (TID) REFERENCES MODELTEMPLATE (NAME) ON DELETE CASCADE
)
```

COMPONENTOF

```
CREATE TABLE COMPONENTOF (
TOWNER VARCHAR (100),
TCOMPONENT VARCHAR (100),
TICOMPONENT INTEGER,
PRIMARY KEY (TOWNER, TCOMPONENT, TICOMPONENT),
FOREIGN KEY (TOWNER) REFERENCES MODELTEMPLATE (NAME) ON DELETE CASCADE,
FOREIGN KEY (TCOMPONENT, TICOMPONENT) REFERENCES MODELTI (TID, TIID) ON DELETE CASCADE
)
```

PORTTI

```
CREATE TABLE PORTTI (
TOWNER VARCHAR (100),
TIOWNER INTEGER,
TNAME VARCHAR (100),
TTYPE VARCHAR (5),
PRIMARY KEY (TOWNER, TIOWNER, TNAME, TTYPE),
FOREIGN KEY (TOWNER, TIOWNER) REFERENCES MODELTI (TID, TIID) ON DELETE CASCADE,
FOREIGN KEY (TOWNER, TNAME, TTYPE) REFERENCES PORTTEMPLATE (OWNER, TNAME, TTYPE) ON
DELETE CASCADE
)
```

COUPLING

```
CREATE TABLE COUPLING (
TOWNERF VARCHAR (100),
TIOWNERF INTEGER,
TNAMEF VARCHAR (100),
TTYPEF VARCHAR (5),
TOWNERT VARCHAR (100),
TIOWNERT INTEGER,
TNAMET VARCHAR (100),
TTYPET VARCHAR (5),
PRIMARY KEY (TOWNERF, TIOWNERF, TNAMEF, TTYPEF, TOWNERT, TIOWNERT, TNAMET, TTYPET),
FOREIGN KEY (TOWNERF, TIOWNERF, TNAMEF, TTYPEF) REFERENCES PORTTI (TOWNER, TIOWNER,
TNAME, TTYPE) ON DELETE CASCADE,
FOREIGN KEY (TOWNERT, TIOWNERT, TNAMET, TTYPET) REFERENCES PORTTI (TOWNER, TIOWNER,
TNAME, TTYPE) ON DELETE CASCADE
)
```

MODELINSTANCE

```
CREATE TABLE MODELINSTANCE (
TEMPLATE VARCHAR (100),
TEMPLATEI INTEGER,
IID INTEGER,
MODELNAME VARCHAR (100),
CREATETIME INTEGER,
PRIMARY KEY (TEMPLATE, TEMPLATEI, IID),
FOREIGN KEY (TEMPLATE, TEMPLATEI) REFERENCES MODELTI (TID, TIID) ON DELETE CASCADE
)
```

MTITOSMI

```
CREATE TABLE MTITOSMI (
TTEMPLATE VARCHAR (100),
TITEMPLATE INTEGER,
TSPECIALIZATION VARCHAR (100),
TISPECIALIZATION INTEGER,
ISPECIALIZATION INTEGER,
PRIMARY KEY (TTEMPLATE, TITEMPLATE, TSPECIALIZATION, TISPECIALIZATION, ISPECIALIZATION),
FOREIGN KEY (TSPECIALIZATION, TISPECIALIZATION, ISPECIALIZATION) REFERENCES MODELINSTANCE
(TEMPLATE, TEMPLATEI, IID) ON DELETE CASCADE,
FOREIGN KEY (TTEMPLATE, TITEMPLATE) REFERENCES MODELTI (TID, TIID) ON DELETE CASCADE
)
```

COMPONENTOFI

```
CREATE TABLE COMPONENTOFI (
TOWNER VARCHAR (100),
TIOWNER INTEGER,
IOWNER INTEGER,
TCOMPONENT VARCHAR (100),
TICOMPONENT INTEGER,
ICOMPONENT INTEGER,
PRIMARY KEY (TOWNER, TIOWNER, IOWNER, TCOMPONENT, TICOMPONENT, ICOMPONENT),
FOREIGN KEY (TOWNER, TIOWNER, IOWNER) REFERENCES MODELINSTANCE (TEMPLATE, TEMPLATEI, IID)
ON DELETE CASCADE,
FOREIGN KEY (TCOMPONENT, TICOMPONENT, ICOMPONENT) REFERENCES MODELINSTANCE (TEMPLATE,
TEMPLATEI, IID) ON DELETE CASCADE
)
```

5.5. SESM Switch Network Example

ModelTemplate		
Name	tModelType	createTime
SN	COUPLED	1384987
PacketSwitch	SPECIALIZED	1938390
IP6 Switch	ATOMIC	1939078
IP5 Switch	ATOMIC	1938983

PortTemplate			
Owner	tName	tType	createTime
SN	Out Port	OUT	1938494
SN	In Port	IN	1930383
PacketSwitch	In 1	IN	1938539
PacketSwitch	In 2	IN	1939303
PacketSwitch	Out 1	OUT	1939133
PacketSwitch	Out 2	OUT	1938974
IP6 Switch	In 1	IN	1938539
IP6 Switch	In 2	IN	1939303
IP6 Switch	Out 1	OUT	1939133
IP6 Switch	Out 2	OUT	1938974
IP5 Switch	In 1	IN	1938539
IP5 Switch	In 2	IN	1939303
IP5 Switch	Out 1	OUT	1939133
IP5 Switch	Out 2	OUT	1938974

modelTI		
tID	tiID	createTime
SN	0	1384987
PacketSwitch	0	1938390
IP6 Switch	0	1939078
IP5 Switch	0	1938983
PacketSwitch	1	1939405
PacketSwitch	3	1939505
PacketSwitch	2	1939465

portTI			
tOwner	tiOwner	tName	tType
SN	0	Out Port	OUT
SN	0	In Port	IN
PacketSwitch	0	In 1	IN
PacketSwitch	0	In 2	IN
PacketSwitch	0	Out 1	OUT
PacketSwitch	0	Out 2	OUT
PacketSwitch	1	In 1	IN
PacketSwitch	1	In 2	IN
PacketSwitch	1	Out 1	OUT
PacketSwitch	1	Out 2	OUT
PacketSwitch	2	In 1	IN
PacketSwitch	2	In 2	IN
PacketSwitch	2	Out 1	OUT
PacketSwitch	2	Out 2	OUT
PacketSwitch	3	In 1	IN
PacketSwitch	3	In 2	IN
PacketSwitch	3	Out 1	OUT
PacketSwitch	3	Out 2	OUT
IP6 Switch	0	In 1	IN
IP6 Switch	0	In 2	IN
IP6 Switch	0	Out 1	OUT
IP6 Switch	0	Out 2	OUT
IP5 Switch	0	In 1	IN
IP5 Switch	0	In 2	IN
IP5 Switch	0	Out 1	OUT
IP5 Switch	0	Out 2	OUT

modelInstance			
template	templateI	iID	modelName
SN	0	1	ACIMS Switch Network
IP5 Switch	0	1	Cisco Switch 1
IP6 Switch	0	1	Cisco IP6 Switch
IP5 Switch	0	2	Cisco Switch 2

specialization	
template	specialization
PacketSwitch	IP5 Switch
PacketSwitch	IP6 Switch

componentOf		
tOwner	tComponent	tiComponent
SN	PacketSwitch	1
SN	PacketSwitch	2

SN	PacketSwitch	3

coupling							
tOwnerF	tiOwnerF	tNameF	tTypeF	tOwnerT	tiOwnerT	tNameT	tTypeT
SN	0	Out Port	OUT	PacketSwitch	2	Out 1	OUT
SN	0	In Port	IN	PacketSwitch	1	In 1	IN
PacketSwitch	1	Out 1	OUT	PacketSwitch	2	In 1	IN
PacketSwitch	2	Out 2	OUT	PacketSwitch	3	In 1	IN
PacketSwitch	3	Out 1	OUT	PacketSwitch	2	In 2	IN
PacketSwitch	3	Out 2	OUT	PacketSwitch	1	In 2	IN

MTItoSMI				
tTemplate	tiTemplate	tSpecialization	tiSpecialization	iSpecialization
PacketSwitch	1	IP5 Switch	0	1
PacketSwitch	2	IP6 Switch	0	1
PacketSwitch	3	IP5 Switch	0	2

componentOfI					
tOwner	tiOwner	iOwner	tComponent	tiComponent	iComponent
SN	0	1	IP5 Switch	0	1
SN	0	1	IP6 Switch	0	1
SN	0	1	IP5 Switch	0	2

5.6. Transaction Requirement & Analysis

SESM environment should provide the user the ability to send commands and queries. In the following, we'll describe the requirements for the SESM to support the necessary transactions given in the Use Case Diagram (see Figure 18).

5.6.1. SESM Manipulation Requirements

Add Template Model – add/create a new Template Model

Add Port – add/create an input port or output port to an existing Template Model

Add Component – Add a Template Model as a component to an atomic Template Model or a coupled Template Model

Add Specialization – add/create a new Template Model as a specialization model specializes an atomic model or a specialized model

Add Coupling – add/create couplings between two ports

Add Instance Model – add Instance Models from a Template Model

Delete Template Model – Delete an existing Template Model

Delete Port – Delete an existing input port or output port from a Template Model

Delete Component – Delete a component from a coupled model

Delete Coupling – Delete a coupling between two ports

Delete Instance Model – Delete an Instance Model and all its components

Modify Template Model Name – Modify a Template Model's name

Modify Instance Model Name – Modify an Instance Model's name

Modify Port Name – Modify a port's name

Modify Model Type – Modify a Template Model's type

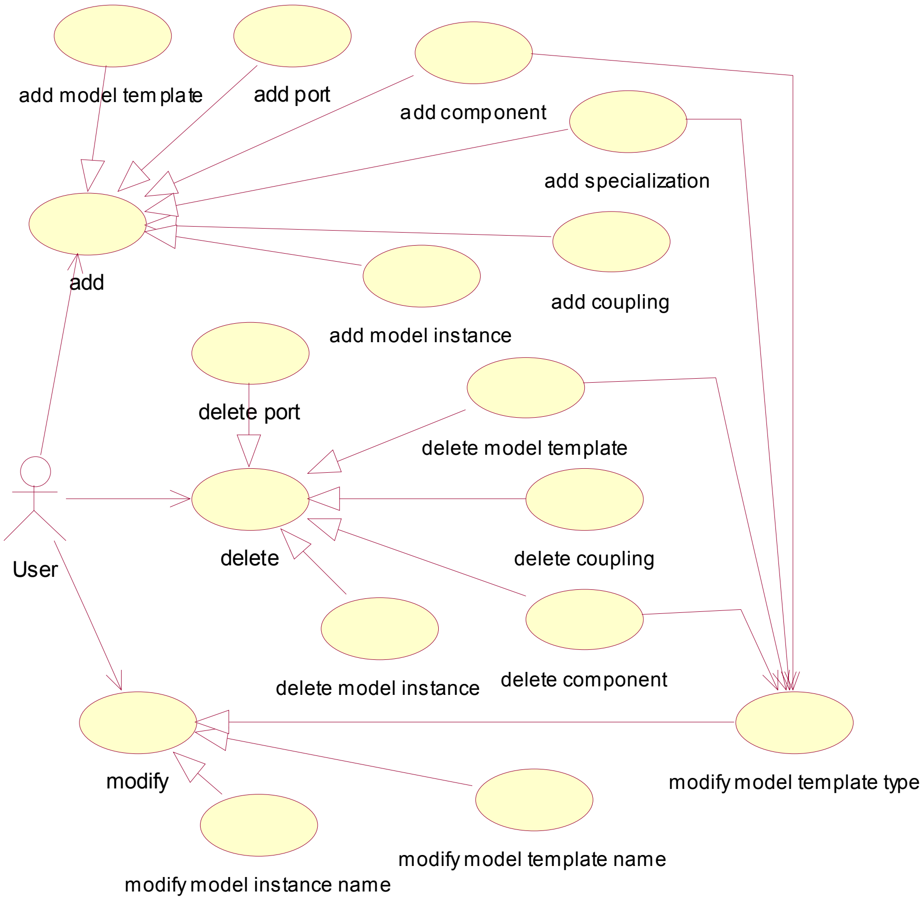


Figure 18: SESM Use Case Diagram

5.6.2. SESM Query Requirements

The required queries can be classified into two categories, Data Query and SESM Model Query. The former handles queries for transaction manipulations described above. The latter is concerned with model re-construction queries. As mentioned in Section 3.4.1, relational models have limitations representing complex structures (e.g., object-oriented style structures). To address such difficulties, the object-oriented structure of SESM was broken into multiple tables as shown in the section 5.4. For instance, a coupled Template Model is broken into modelTemplate, portTemplate, modelTI, portTI, and coupling table. These fragments of information regarding a Template Model are connected by relationships, containsPT, MTtoMTI, PTtoPTI, containsPTI, and componentOf. Thus, for manipulation transaction to modify the correct set of data, it must query the relational database for SESM model data as the transaction progresses into separate tables. As a result, data queries are short transaction queries that are used as parts of some manipulation transaction. Therefore, with SESM queries, users can view models in their object-oriented form. The SESM queries should take into account the GUI to be used since there can be a variety of graphical representations. Further details of SESM model queries are discussed in Chapter 7.

5.6.3. SESM Behavior Requirements

As described in Chapter 3, the RDBMS relational model uses several constraints to maintain its data integrity. The SESM ER diagram tried to implement as many constraints required by DM/SES specification using constraints provided by the relational model. For instance, the specialized relationship can only exist if both the specialized Template Model and specialization Template Model exist. However, because of the RDBMS's limitation on storing behavior as mentioned in section 3.3.1, some of the SESM structural behaviors as stated in Section 5.2 were not expressed in the Figure 17. Thus, behavioral constraints of the SESM model are not implemented in the DBMS. For example, the model behavior that a Template Model and its instances must have the same structure is not enforced by the DBMS. The satisfiability of such behaviors is implemented (using Java programming languages) in various parts of the SESM (e.g., client, server, and network shown in Figure 12). The following lists the requirements that are not satisfied by SESM ER Diagram and the Logical Schema.

Model

- A model cannot be a component of its component (no looping)
- A model that has zero component should be an atomic model
- A model that has one or more components should be a coupled model

Couplings

- A coupling should only link between a coupled model and its component or two components of the same coupled model
- A coupling that couples a coupled model and its component should couple same type of ports

- A coupling that couples two components of the same coupled model should couple different types of ports (i.e., output port to input port)
- A coupling should be between
  - Coupled Model Input – Component Model Input
  - Component Model Output – Component Model Input
  - Component Model Output – Coupled Model Input

Template Model

- The template name should be assigned by the creator (user)
- When a decomposed Template Model is deleted, all its components should be deleted but not the components' templates
- A specialization Template Model should have the exact interface (input ports and output ports) as its specialized Template Model

Instance Template Model

- An Instance Template Model should have the identical structure (components, ports & couplings) as its template
- An Instance Template Model should be a component of at most one other Instance Template Model
- An Instance Template Model should have the identical structure (components, ports & couplings) as its template

Instance Model

- An Instance Model should only be created from either a Coupled Template Model or Atomic Template Model
- An Instance Model should have the identical structure (components, ports & couplings) as its template
- When an instance is deleted, all its components should be deleted
- Specialization does not exist in the instance level. A Specialization Template Model that specializes the original Template Model must be selected when the Instance Model is created.

5.7. SESM Transactions Specification

Data Manipulation Transactions

Due to the fact that SESM model's complex structure is flattened into multiple tables, transactions that are to manipulate the structure must also be applied to multiple tables. A single transaction often requires nested queries and data manipulations to the relational database. This type of transaction is known as the "long transaction". Compared with the "short transaction", an atomic transaction that is supported in relational database and can be expressed in SQL, a long transaction is not supported in the relational database directly. As a result, all SESM data manipulation transactions are broken into short transactions that are supported in the relational database.

The data manipulations transactions can be classified into three categories, add, delete and modify. Each category of transactions is described in details in the following sections. For each transaction, a brief description is given. The input and output values to the transaction and the restriction of the values are described. The short transactions that make up the long transaction are expressed in the format of "Transaction Description (Type of Transaction)". The types of transaction can be SQL INSERT, SQL DELETE, SQL UPDATE, or a CALL. SQL INSERT means that a SQL statement with keyword "INSERT" would be passed to the relational database. The same is applied to SQL DELETE and SQL UPDATE, except the keyword is "DELETE" and "UPDATE". Examples of each SQL statement with the specific key words are shown below.

- SQL INSERT: INSERT INTO MODELTEMPLATE VALUES ('SN', 'ATOMIC', 996040347645)
- SQL DELETE: DELETE PORTTEMPLATE WHERE OWNER='SN' AND TName='Out Port' AND TTYPE='OUT'
- SQL UPDATE: UPDATE MODELTEMPLATE SET TMODELTYPE = 'COUPLED' WHERE TID='SN'
- SQL SELECT: SELECT TMODELTYPE FROM MODELTEMPLATE WHERE TID='SN'

When a SQL level transaction is used, the corresponding table in the relational database will be specified. The CALL means that logic implemented in a programming language (i.e., Java) will be executed.

5.7.1. Add Transactions

In general, add transactions involve both adding entities and relationships. Sometimes add transactions also need to modify the data. For example, consider adding a Template Model "Packet Switch" as a component of Template Model "SN". Briefly, the add transaction includes the creation of "Packet Switch" Instance Template Model and adding the componentOf relationship between "Packet Switch" and "SN", plus modifying the model type of "SN" to "COUPLED". More details of each add transaction is described below.

Add Template Model

Input:

- Template Model name

Restriction:

- The inputted Template Model name cannot exist

Output:

- None

Transactions:

- Store Template Model name as a new atomic Template Model into the modelTemplate table (SQL INSERT on modelTemplate table)

Add Port

Adding an input port or an output port to an existing Template Model. Since it is required for all Instance Models created from the template to maintain the exact model structure, the port added to the Template Model should also be added to all the Template Model's instances and template instances. Furthermore, all specialization Template Models must have the exact interface (input ports and output ports) as its specialized Template Model. As a result, users are only allowed to modify the model's interface through the specialized Template Model. When a port is added to a specialized Template Model, the same port will be added to all its specialization Template Models. Users are not allowed to add or delete a port in a specialization Template Model.

Input:

- Template Model name (owner of the port)
- port name
- port type (‘IN’ or ‘OUT’)

Restriction:

- The inputted port (port name and port type) does not exist in the Template Model

Output:

- None

Short Transactions:

- Add port to the Template Model (SQL INSERT on portTemplate table)
- Check for Template Model’s type and chose from one of the three sets of transactions below
- Atomic Model
- Coupled Model
  - Query for all the Template Model’s Instance Template Models (SQL SELECT on modelTI table)
  - For each Instance Template Model returned from the query, add port to it (SQL INSERT on the portTI table)
- Specialized Model
  - Query for all the Template Models that specialization the specialized model (SQL SELECT on specialization table)
  - For each Template Model returned from the query, add port to it (CALL: Add Port)

Add Component

Input:

- Template Model name (coupled model)
- Template Model name (component)

Restrictions:

- The inputted coupled Template Model cannot be a specialized Template Model
- The added component should not create a cycle in either the coupled Template Model and component Template Model hierarchy or the specialized model and specialization model hierarchy.

Output:

- None

Short Transactions:

- Query for an non-used template instance ID for the component (SQL SELECT on modelTI table)
- Use the template instance ID obtained, create a Instance Template Model from the component's Template Model (SQL INSERT on modelTI table)
- Create the component of relationship of the coupled model and its component. (SQL INSERT on componentOf table)
- Query for all the Instance Models created from the coupled Template Model (SQL SELECT on modelInstance table)
- For each Instance Models returned from the query above, Add an instance of the component Template Model using the Instance Template Model created in the previous step. (CALL: Add Instance Model)
- For each Instance Model created, add the component of instance relationship of the coupled Instance Model and the component Instance Model (SQL INSERT on componentOfI table)
- Change the model type to "COUPLED" (SQL UPDATE on modelTemplate table)

Add specialization

Add specialization transaction creates a new Template Model from the chosen Template Model as its specialization model. As specified in the requirements, the specialization Template Model must have the same interface as the specialized model. As a result, all ports in the specialized model should be added to the specialization model created.

Input: Template Model name (specialized model) and Template Model name (specialization model)

Input:

- Template Model name (Specialized Template Model)
- Template Model name (Specialization Template Model)

Restrictions:

- The specialization Template Model name should not exist
- The specialized Template Model cannot be a coupled Template Model

Output:

- None

Short Transactions:

- Create a new Template Model (CALL: add Template Model)
- Query for all the ports form the specialized Template Model (SQL SELECT on portTemplate table)
- For each port returned, add the port to the new specialization Template Model created (CALL: Add Port)
- Create the specialization relationship (SQL INSERT on specialization table)
- Check the specialized Template Model's model type (SQL SELECT on modelTemplate table)
- Change the model type to "SPECIALIZED" if necessary (SQL UPDATE on modelTemplate table)

Add coupling

Coupling is used to show how output port p1 is connected to output port p5. Couplings can be stored in two different ways since every coupling must have a direction. For example see coupling table 1

coupling table 1							
tOwnerF	*tiOwnerF*	TNameF	tTypeF	*tOwnerT*	*tiOwnerT*	tNameT	tTypeT
PacketSwitch	*2*	IN Port 1	IN	*PacketSwitch*	*1*	Out Port 0	OUT
PacketSwitch	*1*	Out Port 0	OUT	*PacketSwitch*	*2*	IN Port 1	IN

All couplings are stored in the relational database using the rules given below. For couplings between two components, the coupling is stored as shown in coupling table 2 where output port is stored on the left of the input port

coupling table 2							
tOwnerF	*TiOwnerF*	tNameF	tTypeF	*tOwnerT*	*tiOwnerT*	tNameT	tTypeT
PacketSwitch	*1*	Out Port 0	OUT	*PacketSwitch*	*2*	IN Port 1	IN

For couplings between a coupled model and its component, the coupling is stored as shown in the coupling table 3 where coupled model's port is stored on the left of the component's port

coupling table 3							
tOwnerF	*tiOwnerF*	tNameF	tTypeF	*tOwnerT*	*tiOwnerT*	tNameT	tTypeT
SN	*0*	In 0	IN	*PacketSwitch*	*1*	IN Port 1	IN

Input:

- A pair of ports (Template Model name, Instance Template Model ID, port name, and port type)

Restrictions:

- The ports should belong to two component models belong to the same coupled model or a coupled model and its component.
- If both ports belong to two component models, one port should be an input Port and the other should be an output port.
- If the ports belong to coupled model and its component, both ports should has the same port type

Output:

- None

Short Transactions:

- Add the coupling (SQL INSERT on coupling table)

Add Instance Model

When adding an Instance Model, the transformation process is used to convert a Instance Template Model into an Instance Model. When a Template Model is transformed into an Instance Model, the process depends on the type of the Template Model. If the Template Model is an atomic model, the transformation is straightforward, just transforming the Template Model into a Instance Model. If the Template Model is a specialized model, a Template Model that specializes the specialized Template Model must be chosen. If the Template Model is a coupled model, all its components must be transformed into Instance Models recursively.

Input:

- Template Model ID

Output:

- Instance Model ID

Short Transactions:

- Query the Template Model type (SQL SELECT on the modelTemplate table)
- Using one of the transactions below depends on the Template Model type
- Add an Instance Model from an atomic Template Model
- Select a non-used Instance Model ID for the Template Model (SQL SELECT on modelInstance table)
- Transform the Template Model into an Instance Model (SQL INSERT on modelInstance table)
- Add an Instance Model from a coupled Template Model
    - Select a non-used Instance Model ID for the Template Model (SQL SELECT on modelInstance table)
    - Transform the Template Model into a Instance Model (SQL INSERT on modelInstance table)
    - Query for all the coupled model's components (SQL SELECT on componentOf table)
    - For each component returned, add the component as a Instance Model (CALL: Add Instance Model)
- For each component Instance Model added, create component of instance relationship (SQL INSERT on componentOfI table)
- Add an Instance Model from a specialized Template Model

- Query for all the specialized model's components (SQL SELECT on specialization table)
- Ask the user to select one of the specialization models (CALL: request for user input)
- Add the chosen specialization model as an Instance Model (CALL: Add Instance Model)
- Record the purring selection (SQL INSERT on "MTItoSMI" table)

5.7.2. Delete Transactions

The delete transaction often includes both the delete of entities and relations. Although the relational database management system can support cascade delete in order to support delete as a short transaction, cascade delete is not sufficient to support the delete transaction required by SESM. For instance, a component of a coupled model should not exist without the coupled model. Yet, for the same component, its existence also requires the existence of its Template Model. This dual dependence of the component makes it impossible to solely rely on cascade delete to ensure the correctness of the model structure.

Delete Template Model

When a Template Model is deleted, all the entities that associated with it must be deleted to maintain the model structure's correctness. For an atomic Template Model, all its Instance Models should be deleted. For a coupled Template Model, all its components and its Instance Models should be deleted. In other words, all the Instance Template Models that are created as its components should be deleted. For a specialized Template Model, all its specialization Instance Models should be deleted. Thus, the transaction of removing a Template Model depends on it type. After the Template Model has been deleted, all the Template Models used it as a component or as a specialization model should be check and update their type if necessary.

Input:

- Template Model name

Short Transactions:

- Query for the Template Model's type (SQL SELECT on modelTemplate table)
- Query for all coupled Template Models that contain the Template Model as a component (SQL SELECT on componentOf table)

- Query for all specialized Template Models that are specialized by the Template Model (SQL SELECT on specialization table)
- Depends on the Template Model's type, one of the three set of transactions below is chosen
- Delete a atomic Template Model
  - Delete the Template Model (SQL DELETE on modelTemplate table)
- Delete a coupled Template Model
  - Query for the coupled model's component (SQL SELECT on componentOf table)
  - For each component returned, delete the component (SQL DELETE on modelTI table)
- Delete a specialized Template Model
  - Query for the specialization Instance Model created based on the specialized Template Model (SQL SELECT on MTItoSMI table)
  - For each specialization Instance Model returned, delete the Instance Model (CALL: Delete Instance Model)
  - Query for all Instance Models created from the Template Model (SQL SELECT on modelInstance table)
  - For each Instance Model returned, delete the Instance Model (CALL: Delete Instance Model)
  - After all the Instance Models have been deleted, delete the Template Model inputted (SQL DELETE on modelTemplate table)
  - Check for all the coupled Template Models queries earlier and change their type if necessary (SQL UPDATE on modelTemplate table)
  - Check for all the specialized Template Model queries earlier and change their type if necessary (SQL UPDATE on modelTemplate table)

Delete Port

Input:

- Template Model ID
- Port name
- Port type

Restriction

- The inputted port should not belong to a specialization Template Model

Output:

- None

Short Transactions:

- Delete port from the Template Model (SQL DELETE on portTemplate table)
- If the Template Model is specialized execute the following transactions
    - Query for all Template Models that specialization the specialized Template Model. (SQL SELECT on specialization table)
    - For each Template Model returned, delete the port from the Template Model (CALL: Delete Port)

Delete Component

Input:

- Coupled Template Model ID
- Component's Template Model ID
- Component's Instance Template Model ID

Output:

- None

Short Transactions:

- Query for all the instances created from the coupled Template Model (SQL SELECT on modelInstance table)
- For each Instance Model returned, delete the component inputted (CALL: Delete Instance Model)
- Delete the component inputted from the coupled Template Model (SQL DELETE on modelTI table)
- Check the coupled Template Model and update its type if necessary (SQL UPDATE on modelTemplate table)

Delete Coupling

Input:

- A pair of ports (Template Model name, Instance Template Model ID, port name, and port type)

Output:

- None

Short Transactions:

- Delete the coupling (SQL DELETE on coupling table)

Delete Instance Model

Each Instance Model and its components can be seen as a tree where the atomic Instance Models are leaf nodes. The user should only be allowed to delete root Instance Models. In other words, the user should not be allowed to delete Instance Models used as a component of another model. These instances models should be deleted recursively when the Instance Model at the root level is deleted.

Input:

- Template Model name
- Instance Model ID

Restriction:

- The Instance Model inputted should not be a component of any other Instance Models.

Output:

- None

Short Transactions:

- Query for all the Instance Model's component (SQL SELECT on componentOfI table)
- For each Instance Model returned, delete the Instance Model (CALL: Delete Instance Model)

5.7.3. Modify Transactions

Modify transactions are performed to modify the name of an object (Template Model, port, or Instance Model) and the type of a Template Model. In the case where the modifications are performed on attributes that are used as foreign reference (i.e., references to rows in other tables), by other tables, cascade update must be performed. Different from the other types of data manipulation transactions (add and delete), modify does not change the structure of the model.

Therefore, modify transactions do not need extra constraints other than those applied by the RDBMS (e.g., referential integrity constraint). The referential integrity constraint ensures that any foreign reference data is referencing to an existing data. To comply with this constraint, a cascade update, updating all its references when an entity's key value is updated, must be used for modifying a Template Model name and port name since both are used as foreign references at several tables. Depending on the DBMS used, cascade modify might not be supported directly. In such cases, two implementation methods can be used to accomplish the cascade update. The first method is implementing a trigger on the DBMS. The trigger will execute the cascade update on the server side when the specific entity that trigger is assigned is updated. The other method is to perform the updates in the application. The second method is the preferred choice since the desired data is loaded into memory and deleted from the DBMS. Then, the updated data is written back into the DBMS. The method is chosen for the following reasons. First, the size of the data is relatively small compared with the typical size of a database. Secondly, performance is not the primary concern. Third, this method is more portable compared to implementing a trigger in the DBMS.

Modify Template Model Name

Input:

- Template Model name (old)
- Template Model name (new)

Restriction

- The inputted Template Model (new) must be unique to all Template Model names in the database

Output:

- None

Short Transactions:

- Query for all rows contains the Template Model name (SQL SELECT on all tables)
- For each row returned, replace the old Template Model name with the new Template Model name inputted (CALL)
- Write all modified rows back to its original table (SQL INSERT on all tables)

Modify Instance Model Name

Input:

- Template Model name
- Instance Model ID
- Instance Model name (new)

Output:

- None

Short Transactions:

- Modify the Instance Model's name (SQL UPDATE on modelInstance table)

Modify Port Name

Input:

- Template Model name
- Port name (old)
- Port type
- Port name (new)

Output:

- None

Restriction

- The inputted port name (new) must be unique to all port in the Template Model with the same type

Short Transactions:

- Query for all rows that contain the port inputted (SQL SELECT on portTemplate, portTI, and coupling tables)
- For each row returned, replace the old port name with the new port name inputted
- For each row returned, delete the row from the database (SQL DELET on portTemplate, portTI, and coupling tables)
- Write all rows modified back to its original table (SQL INSERT on portTemplate, portTI, and coupling tables)
- If the Template Model is a specialized Template Model, execute the following transactions (SQL SELECT on modelTemplate table)

- Query for all specialization Template Model that specialization the inputted Template Model (SQL SELECT on specialization table)
- For each Template Model returned, change the port name (CALL: Modify Port Name)

Modify Template Model Type

Modify Template Model type transaction should not be accessed by the user directly. It should only be used to aid add transactions to change the Template Model type internally.

Input:

- Template Model name
- Model type (new)

Output:

- None

Short Transactions:

- Modify the Template Model's type (SQL UPDATE: update on modelTemplate table)

5.7.4. Data Query

Data query is a query needed to support model manipulation transactions. These transactions use these data queries to retrieve information regarding a model that is divided and stored in several tables. Given the design of the manipulation transactions, the data queries are often performed on a single set of data (a single table) and no sorting is required. These queries are therefore SQL SELECT statements executed on a single table. Due to the fact that a wide variety of data queries with different conditions and return data are needed, they are not listed individually here.

6. SESM SERVER ANALYSIS AND DESIGN

6.1. SESM Server Analysis

The SESM defined in the previous chapter allows a user to perform add, delete and modify operations to the models (see Figure 19). These operations involve multiple transactions to the relational database. Figure 20 shows package diagram of SESM Server The Figure 21 shows the relationships between SESM transactions (Add, Delete and Modify) and SQL transactions (add a row to a table, delete a row from a table, etc). The use-case diagram also represents that the server (see Figure 19) should have the knowledge of relational database schema and the ability to initialize the relational database.

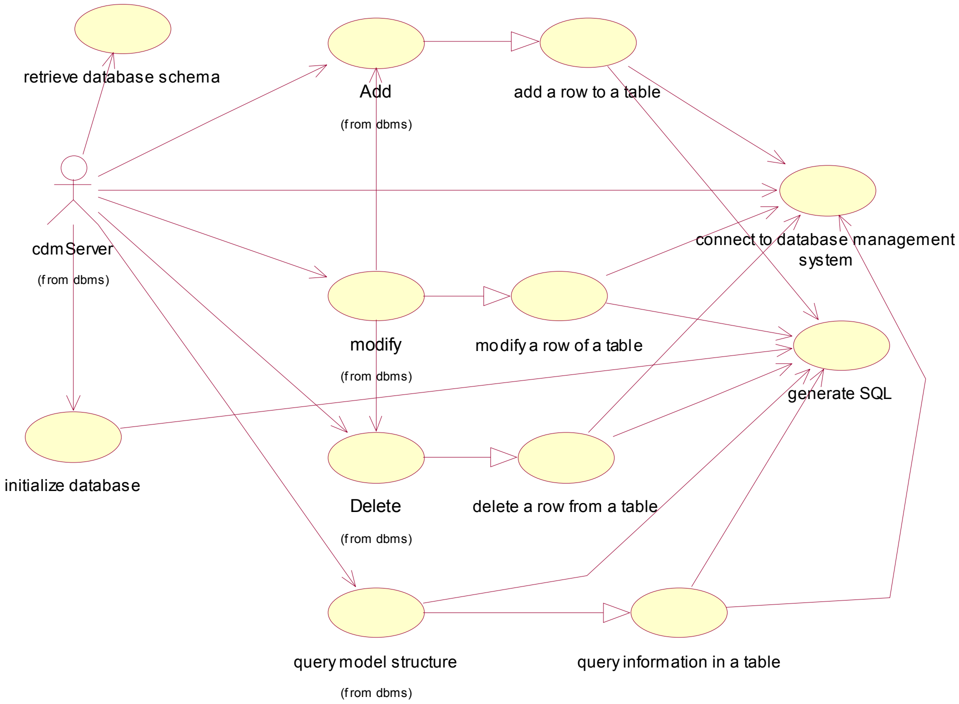


Figure 19: SESM Server Use Case Diagram

6.1.1. SESM Server Integration with Network Environment

Based on the architecture of the SESM, the Server should be loosely coupled with the GUI with the network environment. The Server, therefore, should extend the component provided by the network Environment by adding the logic to process messages and modify the database accordingly. Furthermore, the Server should support network behavior (e.g., correct ordering of transactions to the database) such as sending notification and requesting user input see Figure 16. A class of sesmEvent is also defined and implemented to group data into a single object

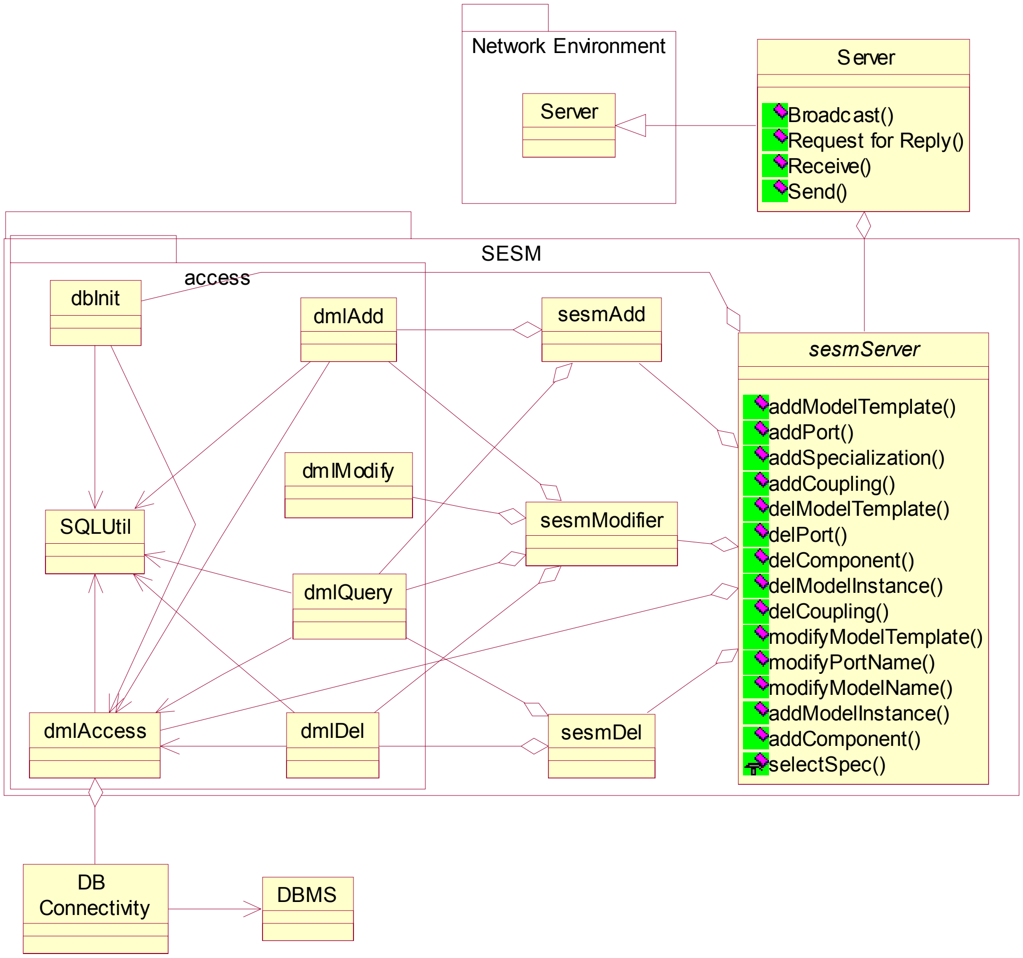


Figure 20 SESM Server Component Diagram

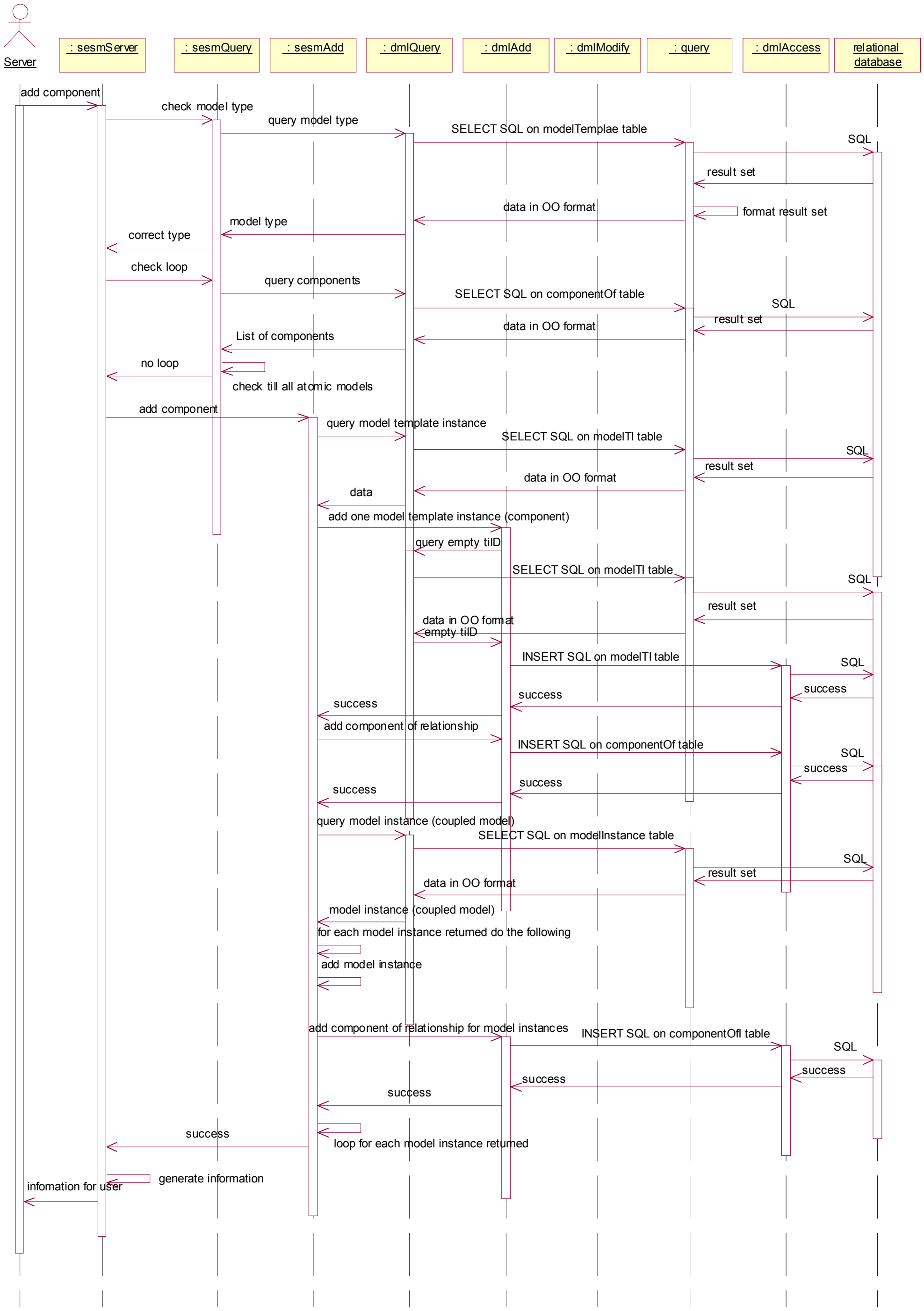

Server
: sesmServer
: sesmQuery
: sesmAdd
: dmlQuery
: dmlAdd
: dmlModify
: query
: dmlAccess
relational database
add component
check model type
query model type
SELECT SQL on modelTemplae table
SQL
result set
data in OO format
format result set
model type
correct type
check loop
query components
SELECT SQL on componentOf table
SQL
result set
data in OO format
List of components
no loop
check till all atomic models
add component
query model template instance
SELECT SQL on modelTI table
SQL
result set
data in OO format
data
add one model template instance (component)
query empty tiID
SELECT SQL on modelTI table
SQL
result set
data in OO format
empty tiID
INSERT SQL on modelTI table
SQL
success
success
success
add component of relationship
INSERT SQL on componentOf table
SQL
success
success
success
query model instance (coupled model)
SELECT SQL on modelInstance table
SQL
result set
data in OO format
model instance (coupled model)
for each model instance returned do the following
add model instance
add component of relationship for model instances
INSERT SQL on componentOfI table
SQL
success
success
success
success
loop for each model instance returned
generate information
infomation for user

Figure 21 SESM Server Sequence Diagram

6.2. SESM Server Design

6.2.1. SESM Package Overview

The SESM package is classified into two sub-packages (the SESM and the SESM.access package) and the Server object. The SESM.access, a sub-package of the SESM, provides methods that are mapped from atomic transactions also known as a relational database transaction. The SESM schema (see Section 5.4) is used as specification for these transactions. Based on the schema, the transactions are translated into SQL statements. These SQL statements are executed by the RDBMS via Database Connectivity. Other than short transactions, the SESM.access package provides the connectivity to the relational database and transaction management to ensure the atomicity of the SESM transactions.

The SESM package provides methods that are mapped directly from the transactions specified in Section 5.7, the Transactions Specifications & Design. Since all short transactions are implemented in the SESM.access package, objects in SESM package should have no direct access to the relational database. Thus, the SESM package focuses on implementing the behaviors of the SESM. The SESM package also provides the logics allowing a network environment to integrate with the Server. The interface should have all methods defined in the Server specification (see Section 6.1) plus required communication methods.

Overall, SESM.access is a utility package that supports all short transactions and tools required to interface with a relational database. Objects in the SESM package implement the behavior of the SESM model and SESM transactions as specified in Section 5.7, the Transactions Specifications.

6.2.2. SESM.accessPackage

This package provides objects (dbInit, dmlAccess, dmlAdd, dmlDel, dmlModify, and dmlQuery, query, and SQLUtil), with methods to manipulate and query the relational database. The behaviors of the SESM model are not implemented since schemas are for the relational database. Therefore, all transactions included in the SESM.access package are short transactions expressed in SQL. The SQLUtil object provides static methods to generate SQL statements from generic variables. The dbInit object contains static variables mapped from the SESM database schema defined in the section 5.4 and provides the method to initialize the relational database by outputting CREATE TABLE and DROP TABLE SQL statements (DDL) to the relational database. The dmlAccess object provides connectivity between the application and the RDBMS. Other than controlling a connection between the application and the relational database, this object also manages all transactions to enforce atomicity. All SQL statements that do not return any values are passed to the relational database thru a dmlAccess object. The query object, a specialized dmlAccess object, provides methods specifically designed for handling the returned results from the RDBMS. The rest of the objects, dmlAdd, dmlDel, dmlModify, and dmlQuery are designed based on the four sets of transactions performed on the database; add, delete, modify, and query. The dmlAdd object provides methods to add transactions. The methods output an INSERT SQL statement to the relational database to perform the transaction. There is a method to add a new row to each table defined in the SESM database schema. The dmlDel object provides methods to delete transactions. Each method outputs a DELETE SQL statement to the relational database. There is a method to delete rows from each table defined in the SESMdatabase schema. As defined in the dbInit object, all delete transactions performed are cascade delete based on the foreign constraint. The dmlModify object provides methods to modify transactions. This method generates an output (a MODIFY SQL statement to the relational database) to perform the transaction. The dmlQuery object provides methods to query transactions. This method generates an output (an SELECT SQL statement to the relational database) to perform the transaction. There should also be a method to query each table defined in the SESM database schema.

The SQLUtil object

Attributes:

- None

Methods:

- dropTable: generate SQL DROP TABLE (DML) statement to drop a table
- createTable: generate SQL CREATE TABLE (DDL) statement to create a table
- insert: generate SQL INSERT (DML) statement to insert a new row into a table
- delete: generate SQL DELETE (DML) statement to delete a row from a table
- update: generate SQL UPDATE (DML) statement to update rows in a table
- query: generate SQL SELECT (DML) statement to query the database

The dmlAccess object

Attributes

- userID: the user name for the relational database
- password: the password for the relational database
- ip: the ip address of the relational database system
- dbID: the identification of the relational database management system
- dbConnect: the JDBC connection provided by the relational database management system vendor.

Methods:

- open: open a connection to the relational database
- close: close the current opened connection
- exeSQL: execute a SQL statement (perform an atomic transaction) without return values
- connectionInfo: get the current connection information
- checkConnection: check to see if a connection is opened
- startTransaction: start a long transaction
- endTransaction: end a long transaction

The dbInit object

Attributes:

- SESM Database Schema: the SESM database schema defined is mapped to static variables of the dbInit object.
- dbConnect: connectivity to the relational database.

Methods:

- initDB: initialize the relational database currently connecting to. All the previous data in the database is erased. The schema is then defined in the relational database.

The dmlAdd object

Attributes:

- dbConnect: connectivity to the relational database.

Methods:

- modelT: insert a new row into the modelTemplate table

- modelTI: insert a new row into the modelTI table
- port: insert a new row into the portTemplate table
- portTI: insert a new row into the portTI table
- compomentOf: insert a new row into the componentOf table
- coupling: insert a new row into the coupling table
- modelI: insert a new row into the modelInstance table
- MTItoSMI: insert a new row into the MTItoSMI table
- componentOfI: insert a new row into the componentOfI table
- addRow: insert a new row to a particular table

The dmlDel object

Attributes:

- dbConnect: connectivity to the relational database.

Methods:

- modelT: delete a row in the modelTemplate table
- modelTI: delete a row in the modelTI table
- port: delete a row in the portTemplate table
- portTI: delete a row in the portTI table
- compomentOf: delete a row in the componentOf table
- coupling: delete a row in the coupling table
- modelI: delete a row in the modelInstance table
- MTItoSMI: delete a row in the MTItoSMI table
- componentOfI: delete a row in the componentOfI table
- deleteRow: delete a row in a particular table

The dmlModify object

Attributes:

- dbConnect: connectivity to the relational database.

Methods:

- modelName: modify the model template name (Name) in the modelTemplate table
- modelType: modify the model template type (modelType) in the modelTemplate table

- instanceName: modify the model instance name (modelName) in the modelInstance table
- modifyRow: modify a row in a particular table

The dmlQuery object

Attributes:

- dbConnect: connectivity to the relational database.

Methods:

- modelT: query the modelTemplate table
- modelTI: query the modelTI table
- port: query the portTemplate table
- portTI: query the portTI table
- componentOf: query the componentOf table
- coupling: query the coupling table
- modelI: query the modelInstance table
- MTItoSMI: query the MTItoSMI table
- componentOfI: query the componentOfI table
- getData: query a particular table

6.2.3. SESM Package

sesmDB

The sesmDB object provides all the operations required to manipulate the database according to the SESM. This object is the interface between all other objects in the SESM package and objects in other packages, like network or GUI. The primary functions of this object are checking correctness of the input and interface with users.

Attributes:

- theQuery: SESM.sesmQuery object used to perform queries retrieving SESM model information
- add: SESM.sesmAdd object used to perform add operations in SESM models
- delete: SESM.sesmDel object used to perform delete operations on SESM models
- modify: SESM.sesmModify object used to perform modify operations in SESM models

Methods:

- addModelTemplate: add a new model template to the relational database
- addPort: add a port to an existing model
- addComponent: add a component to an existing model
- addSpecialization: create a specializing model from an existing model
- addCoupling: couple two existing ports
- addModelInstance: create a new model instance from a model template
- delModelTemplate: delete an existing model template
- delPort: delete a port from a model template
- delComponent: delete a component from a model template
- delCoupling: delete an existing coupling
- delModelInstance: delete an existing model instance at the root level
- modifyModelTemplate: modify the name of an existing model template
- modifyPortName: modify the name of a port
- modifyModelName: modify the name of an existing model instance
- requestUserInput: request the user to select a specializing model for a specialized model. Abstract method should be implemented by the collaborative environment

sesmAdd

The sesmAdd object extends the SESM.access.dmlAdd object to provide add operations. The object uses SESM.access.dmlQuery to gather required information that is used to determine the required add transactions to be performed.

Attributes:

- theQuery: SESM.access.dmlQuery object used to query for model information

Methods:

- modelT: add a new model template to the relational database
- port: add a port to an existing model
- compomentOf: add a component to an existing model
- specialization: create a specializing model from an existing model
- coupling: couple two existing ports

- instance: create a new model instance from a model template

sesmDel

The sesmDel object extends the SESM.access.dmlDel object to provide delete operations. The object uses SESM.access .dmlQuery to gather required information that is used to determine the required delete transactions to be performed.

Attributes:

- theQuery: SESM.access.dmlQuery object used to query for model information

Methods:

- delModelTemplate: delete an existing model template
- delPort: delete a port from a model template
- delComponent: delete a component from a model template
- delCoupling: delete an existing coupling
- delModelInstance: delete an existing model instance at the root level

sesmModify

The sesmModify object extends the SESM.access.dmlModify object to provide modify operations. The object uses SESM.access.dmlQuery to gather required information that is necessary to determine the required add and delete transactions to be performed. The add transactions and delete transactions are carried out by dmlAdd and dmlDel objects.

Attributes:

- theQuery: SESM.access.dmlQuery object used to query for model information
- add: SESM.access.dmlAdd object used to restore the modified values
- delete: SESM.access.dmlDel object used to delete the old values

Methods:

- modifyModelTemplate: modify the name of an existing model template
- modifyPortName: modify the name of a port
- modifyModelName: modify the name of an existing model instance

7. SESM CLIENT ANALYSIS & DESIGN

In the preceding chapters, we discussed SESM Client's architecture and its main three components (see Figure 22). The Client contains graphical user interface and the logics to support user interactions (send and receive) with the Server via the network environment. As shown in Figure 23, the GUI Class utilizes the SESM Query to retrieve data from the DBMS and map the relational models into their object-oriented counterparts. The Figure 23 also shows that the Client can either receive a notification or a request for reply from the Server. If a notification is received, the Client refreshes the GUI accordingly and notifies the user about the modifications made. If a request for reply from the Server was received, the Client interacts with the user to obtain the reply and send it back to the Server. Additionally, the Client is responsible for processing modifications to the models. Upon receiving a modify command, the Client creates a non-block message and sends it to the Server. Although the Client handles all communications between the Server and itself, the Client does not directly access the DBMS since only the GUI is to access the DBMS. Details regarding the GUI are discussed in the next section.

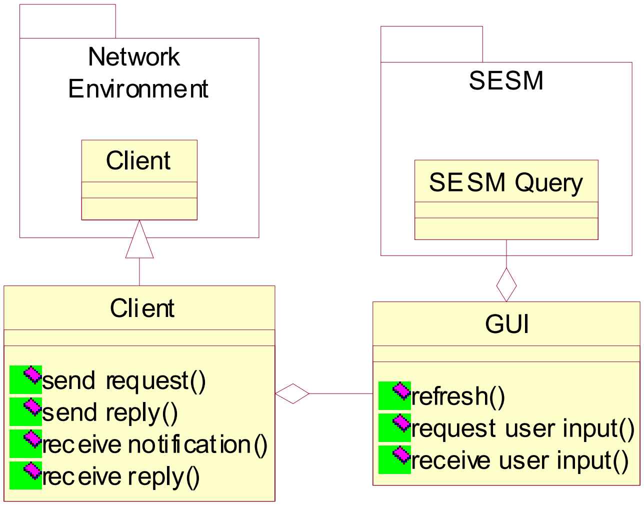


Figure 22 Client Overall Class Diagram

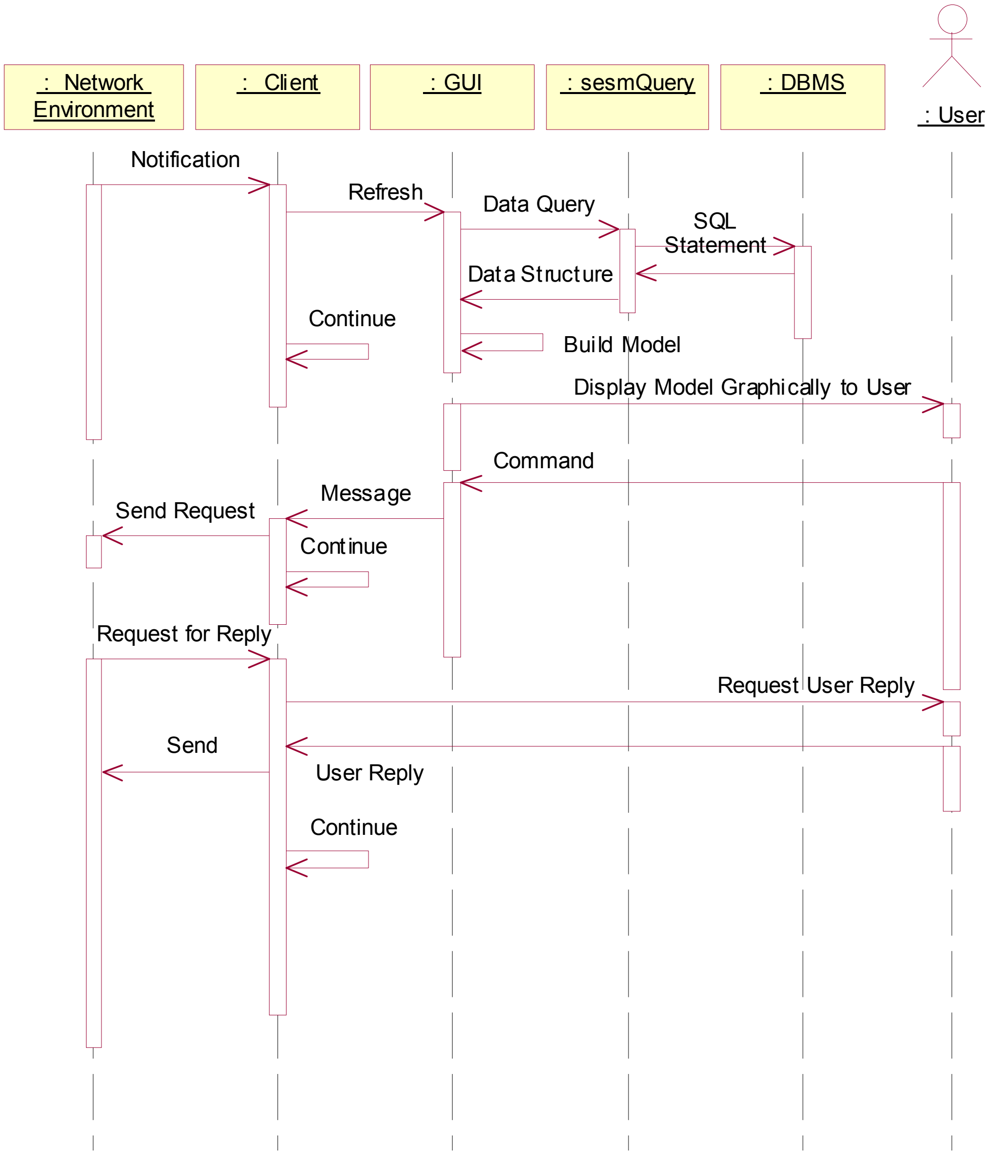


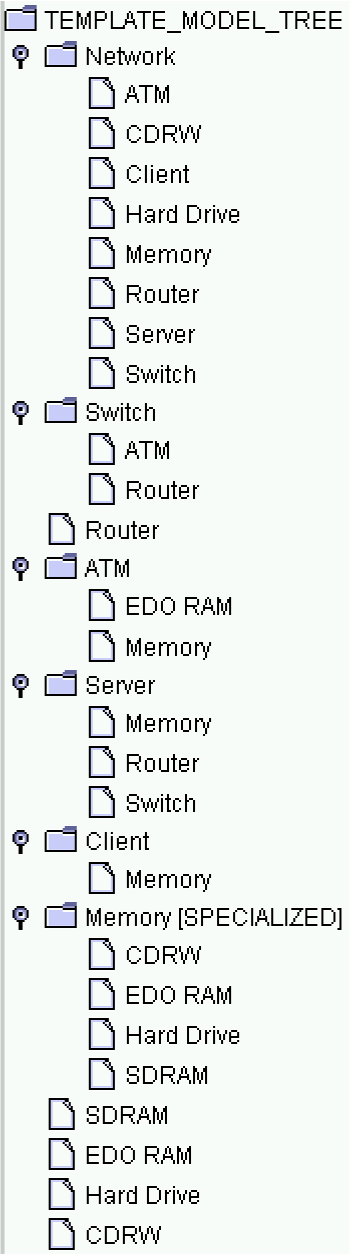
Figure 23 Client Sequence Diagram

7.1. GUI Analysis

The graphical user interface (GUI) has three major functionalities: displaying the hierarchical structure of models, displaying details of models, and allowing the user to modify models. As we stated earlier, a labeled tree can be used to display different model structures. The hierarchical tree structure and details of its models may be displayed in details in a visual model display (VMD). The VMD (See Figure 28 and Figure 29) displays hierarchical representation of models and their detailed specifications (e.g., ports and couplings). The first part (Tree View) can display one of three hierarchical trees (Template Model, Instance Template Model, and Instance Model). The second part (Detailed View) displays the details of a model shown in its Tree View (model decomposition in terms of components, ports, and couplings). The display of ports and components are ordered by their creation-time. The VMD enables a user to create/modify models. The further specification of the integration of the actions and the visualization is discussed in the section 7.1.3.

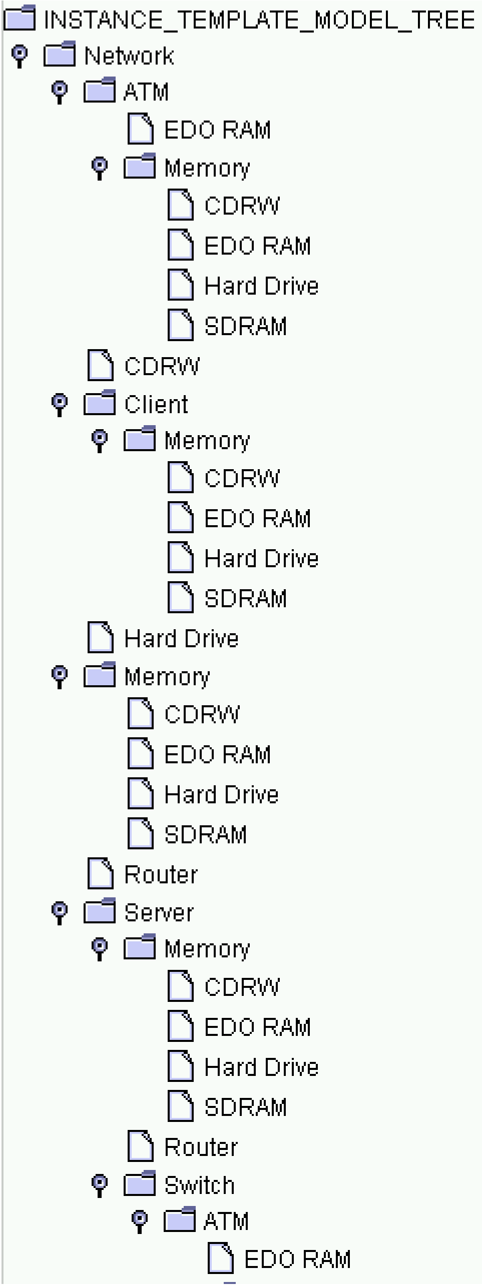
7.1.1. Visual Model Data

The Tree View visualization should represent template model, instance template model, and instance model. The template model tree displays the template model itself and its components or specialized models. Multiple appearances of the same template model is not displayed at this level. In the instance template model tree, the entire hierarchy of the instance template model is displayed. Multiple appearances of the same instance template model are not displayed in this level either. The instance model tree displays the entire hierarchy and multiple appearances of instance models. At this level, the instance models are shown to the user by their instance model name in conjunction with their template model name. Models are displayed in the order in which they are created. To differentiate the specialized model and coupled model, the keyword [SPECIALIZED] is used with the name of the specialized model. Figure 24 shows the GUI display for the Network model. The model depicted in Figure 24 is for illustration purposes and does not represent a part of an actual network system (e.g., the composition relationship between Server, Switch, and Router does not correspond to existing switches).



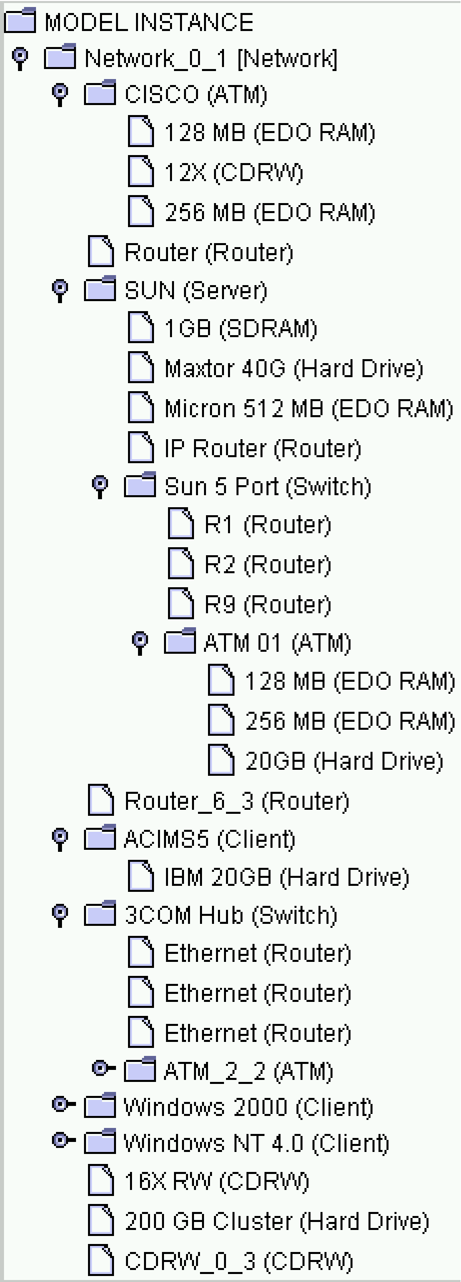
- TEMPLATE_MODEL_TREE
  - Network
    - ATM
    - CDRW
    - Client
    - Hard Drive
    - Memory
    - Router
    - Server
    - Switch
  - Switch
    - ATM
    - Router
  - Router
  - ATM
    - EDO RAM
    - Memory
  - Server
    - Memory
    - Router
    - Switch
  - Client
    - Memory
  - Memory [SPECIALIZED]
    - CDRW
    - EDO RAM
    - Hard Drive
    - SDRAM
  - SDRAM
  - EDO RAM
  - Hard Drive
  - CDRW

Template Model Tree



- INSTANCE_TEMPLATE_MODEL_TREE
  - Network
    - ATM
      - EDO RAM
      - Memory
        - CDRW
        - EDO RAM
        - Hard Drive
        - SDRAM
    - CDRW
    - Client
      - Memory
        - CDRW
        - EDO RAM
        - Hard Drive
        - SDRAM
    - Hard Drive
    - Memory
      - CDRW
      - EDO RAM
      - Hard Drive
      - SDRAM
    - Router
    - Server
      - Memory
        - CDRW
        - EDO RAM
        - Hard Drive
        - SDRAM
      - Router
      - Switch
        - ATM
          - EDO RAM

Instance Template Model Tree



```
MODEL INSTANCE
  Network_0_1 [Network]
    CISCO (ATM)
      128 MB (EDO RAM)
      12X (CDRW)
      256 MB (EDO RAM)
    Router (Router)
    SUN (Server)
      1GB (SDRAM)
      Maxtor 40G (Hard Drive)
      Micron 512 MB (EDO RAM)
      IP Router (Router)
      Sun 5 Port (Switch)
        R1 (Router)
        R2 (Router)
        R9 (Router)
        ATM 01 (ATM)
          128 MB (EDO RAM)
          256 MB (EDO RAM)
          20GB (Hard Drive)
    Router_6_3 (Router)
    ACIMS5 (Client)
      IBM 20GB (Hard Drive)
    3COM Hub (Switch)
      Ethernet (Router)
      Ethernet (Router)
      Ethernet (Router)
      ATM_2_2 (ATM)
    Windows 2000 (Client)
    Windows NT 4.0 (Client)
    16X RW (CDRW)
    200 GB Cluster (Hard Drive)
    CDRW_0_3 (CDRW)
```

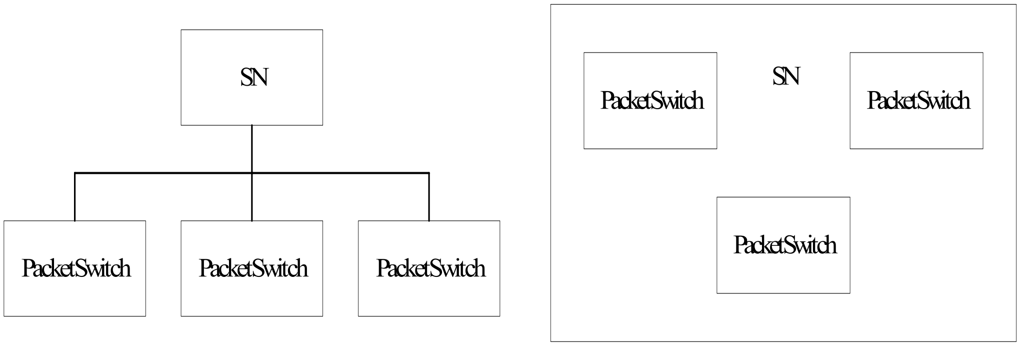
Instance Model Tree

Figure 24: Three Aspects of Model Screen

7.1.2. Visualization of Model

The main goal of model visualization is to create diagrams of a model structure, ports, couplings, and components that are easy to read and manipulate. When visualized, each piece of a model is transformed into a visual object. The combination of these visual objects results in a model's visual diagram. However, many factors (e.g., placement of visual objects, coloring scheme, shape of objects, …) must be taken into account in designing an intuitive and simple visual representation of models. To develop a suitable visualization scheme, it is important to minimize overlapping of objects and enforce the uniformity of the visual objects [Kam 91]. Overlapped objects make the drawing confusing and difficult to read because information is shown partially. Uniformity makes the drawing easy to read for the reason that the same object is drawn the same way irrespective of its placement in a given diagram. This visual uniformity can also be extended to a set of objects where the same type of objects is drawn similarly. For instance, input ports should be drawn the same shape using the same color. The concept of uniformity allows the user to quickly identify the object visually based on its size and colors.

The described concepts and requirements can be realized by applying the constraint-based drawing method [Kam 91]. Each visual object, model, port, and coupling, is drawn according to its set of functions based on the object type. The final size and location of the object is determined by solving the functions using the information represented by the object as variables. The constraint-based drawing eliminates the overlapping of objects and creates visualization uniformity. The constraint-based drawing can also be extended to support hierarchical drawing. In a hierarchical drawing, also known as compound digraph drawing, elements of the graph maintains a hierarchical structure [Kam 91][Sug 91]. By dividing the graph into parts according to the hierarchical structure, the final layout can be built by first constructing each part (sub-graph) and then combining them [Kam 91]. Since SESM structure is strictly hierarchical, the constraint-based drawing method nicely matches it. For example, the SwitchNetwork (SN) can be viewed as a compound graph and hierarchical tree where each PacketSwitch entity is treated as a sub-graph, see Figure 25

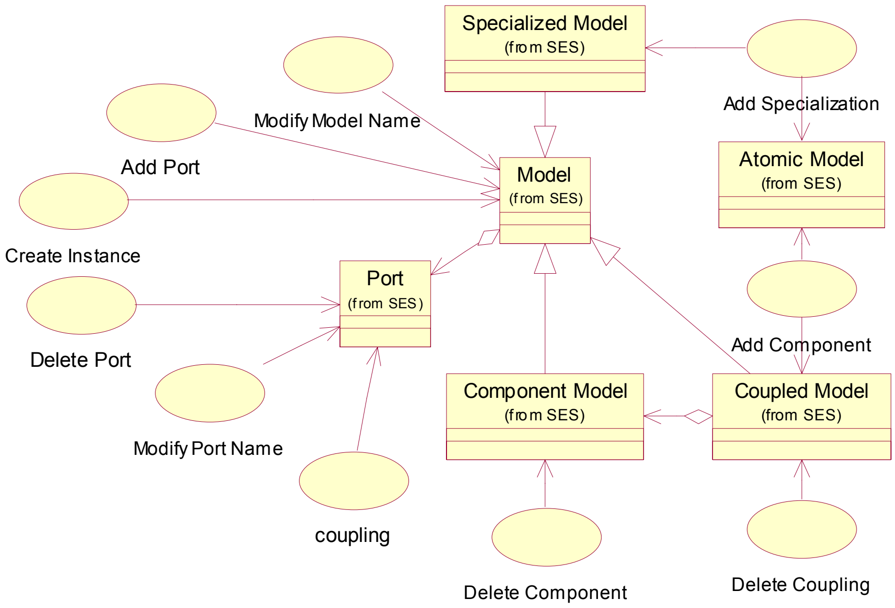


Hierarchical Tree Compound Graph

Figure 25 SwitchNetwork Visual Graphs

7.1.3. Visual Object Based Command Menu

In order to provide simple ways for the user to interact with VMC, each mouse click prompts the user to select a command with a pop-up menu. Each menu is associated with a customized visual object. For instance, the pop-up menu of a port should list the following commands: delete, rename, and coupling. Different types of object have different list of associated commands according to the requirements and transactions defined earlier. By associating a command menu with a specific visual object, the user has a clear view of the commands that may be used. Figure 26 shows commands and their relationships to their corresponding visual objects.



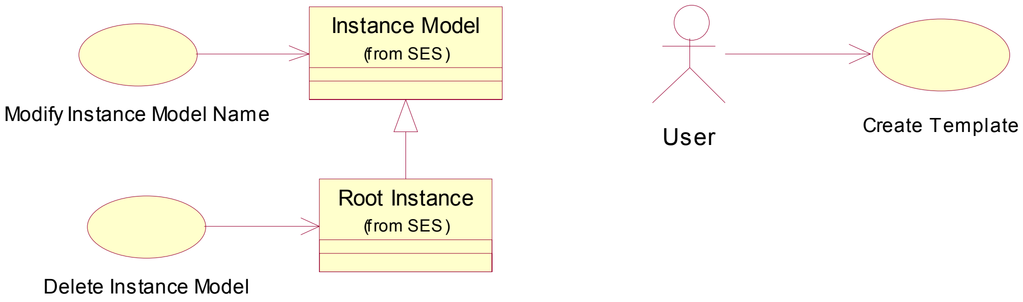


Figure 26: Visual Object Use Case Diagram

7.2. GUI-based Model Retrieval from DBMS

To retrieve the model information from a relational database, GUI uses the queries provided by the SESM package. As we stated earlier, SESM models (e.g., a coupled model) should be presented to users as objects instead of a collection of tables. This is necessary since template, instance template, and instance models are related to one another. For example, the modelTI table by itself does not contain the model type. As a result, one must join modelTemplate table and modelTI table to retrieve complete information regarding an instance template model. Another need for joining tables is to retrieve ports and couplings information for an instance model since ports and couplings of a model are stored in its template model. The Model Queries can be classified into two categories, model data and model hierarchical structure queries. Each Model Query is broken into several SQL SELECT statements. Each SQL SELECT statement is implemented in the SESM package since these statements are completely SQL and based only on the relational database schema. The computational logic, also known as the behavior, of retrieving the relevant data is programmed within the GUI [Sug 91]. Detailed description of data and structure queries are provided in [Fu, 01].

7.3. GUI Design

The UI package is designed to support displaying a set of objects as shown in Figure 27. UI package uses UI.menu and UI.graphics sub-packages to support visual object based command menu and constraint-based drawing respectively. The GUI is implemented using the Swing and

AWT packages in the Java Foundation Classes (JFC). More detailed descriptions of each package are given below.

7.3.1. UI Package

The *UI* package contains *GUI*, *rootModel*, *treeBuilder*, *modelNode*, and *message* classes (Figure 27). The *GUI* object represents the overall GUI that user will interact with. The *rootModel* class provides basic capabilities such as refresh and mode. The treeBuilder class is responsible for creating the three aspects of the model data as trees. The modelNode class handles each component (atomic or coupled model) of the tree structures. The message class provides non-interrupting update message regarding transactions performed on the model data.

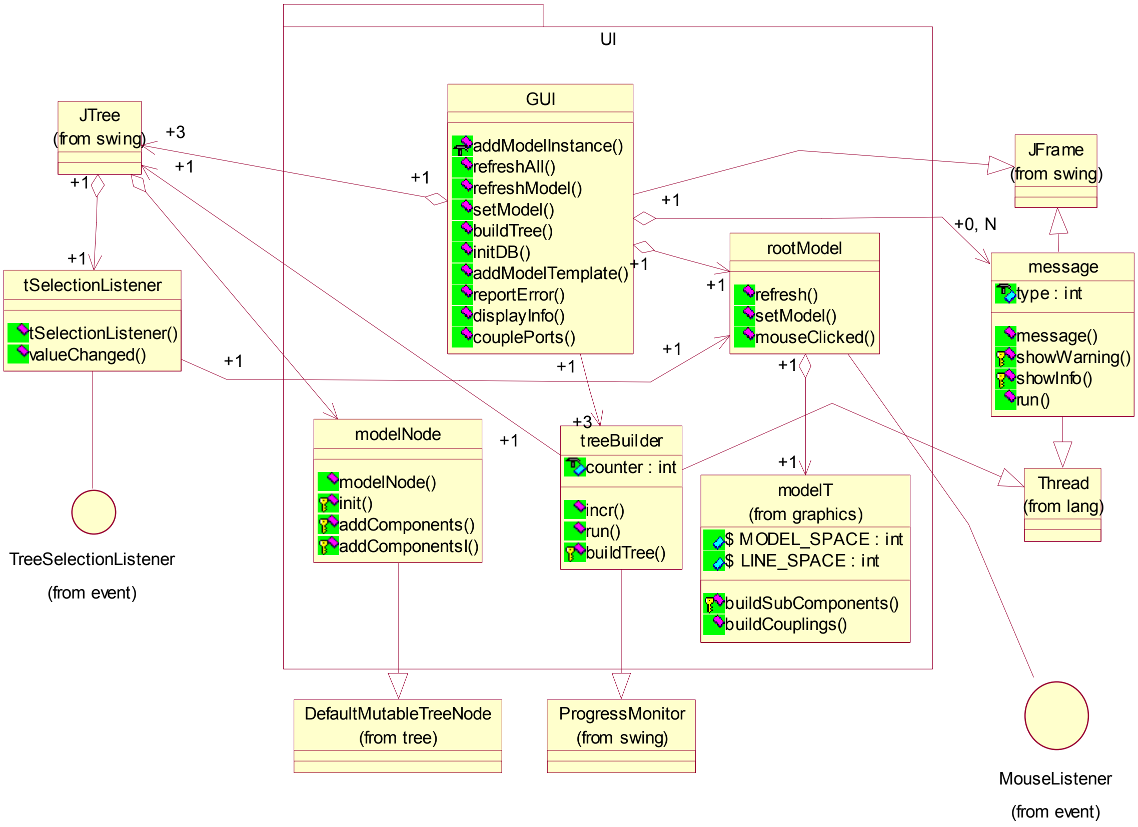


Figure 27: UI Class Diagram

Some of the key user operations are selecting a specific model (e.g., instance model), creating "New Template model", "Create A Instance model", "Initialize Database", and "Refresh" which refreshes the GUI upon user's request.

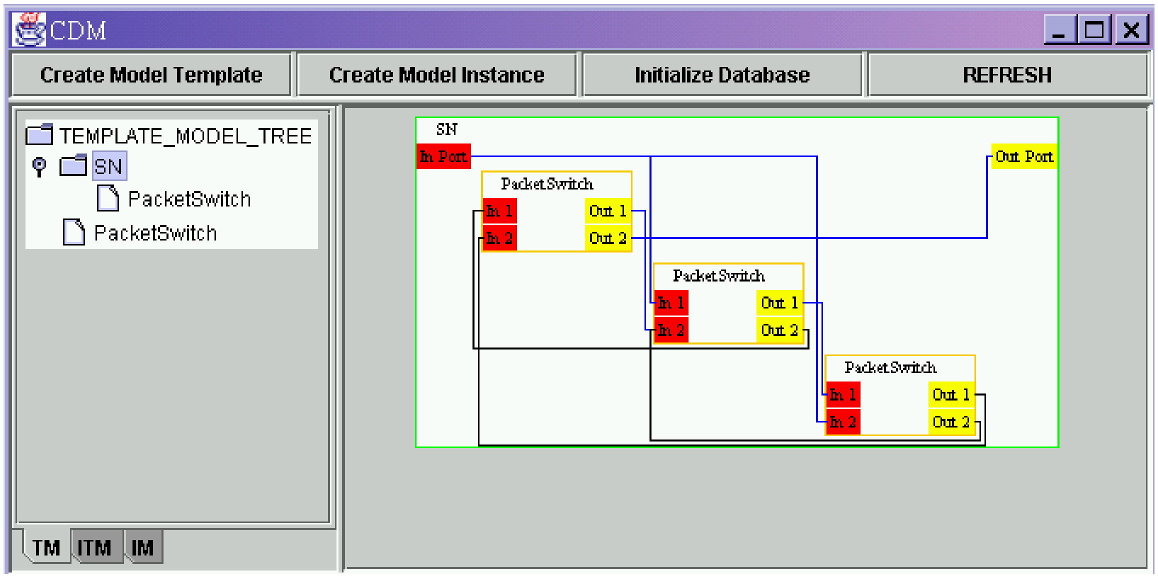


Figure 28: GUI Template Model Screen

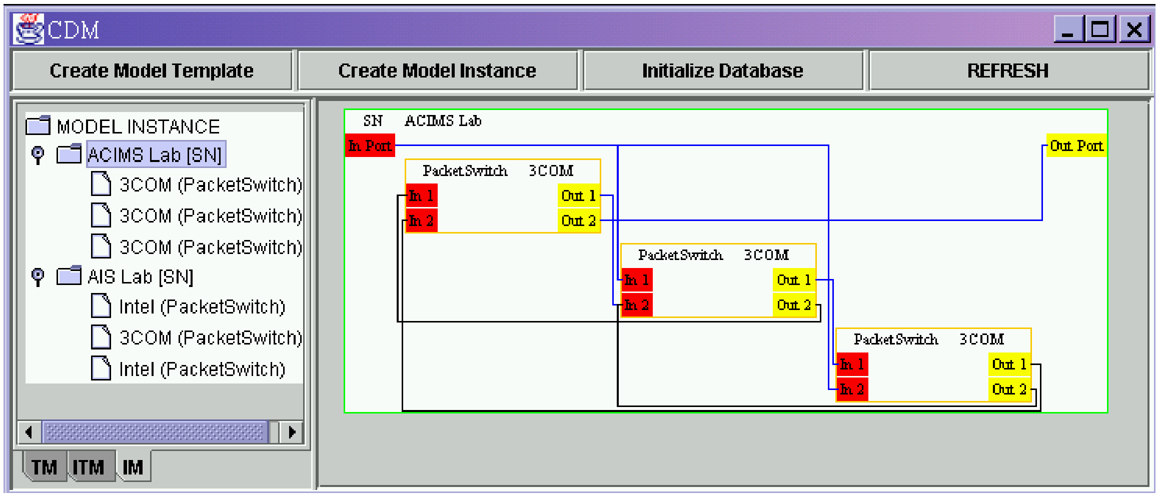


Figure 29: GUI Instance Model Screen

7.3.2. UI.graphics Package

The UI.graphics package consists of a set of classes for visualizing a single or multi-level model (see Figure 30). These classes collectively define the set of constraints (e.g., diagonal placement of model components) that are necessary for organizing components in a systematic fashion (e.g., diagonal placement of components of a coupled model) [Fu 01]. The absGraphics class is the fundamental class for all the other objects in the package. The textArea class is used for visualizing a text string. The port class is for displaying port. A model without any model as its component is visualized by the model class. Different types of models (i.e., atomic and coupled) are shown using two distinct colors. The coupling class is for displaying couplings. The modelT class extends the model object to represent a coupled model in the DM/SES specification. The modelT class is utilized by the GUI object to visualize any model and should be the only object needed to visualize model in a two-level, coupled model and its component, view. The relationships between these objects are shown in the UI.graphics Class Diagram and more details of each object are given below.

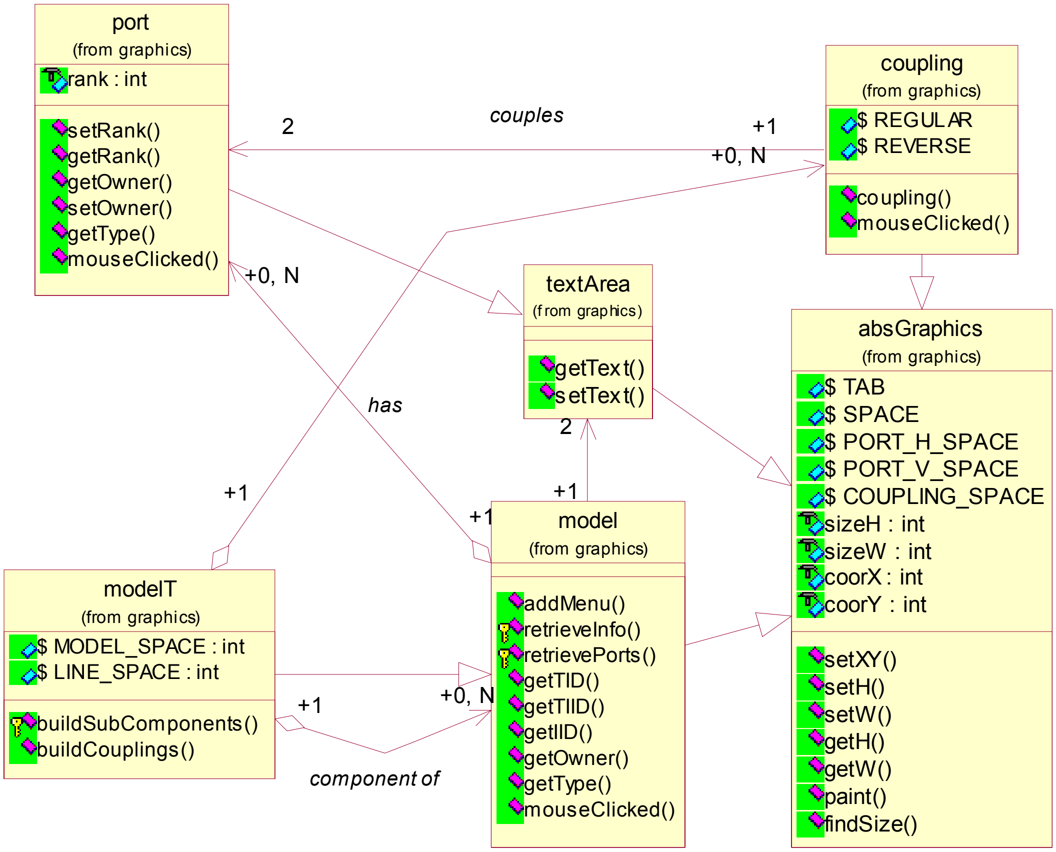


Figure 30: UI.graphics Class Diagram

The UI.graphics package provides event handlings (mouse click) for the classes in the UI.menu package. The interaction between objects in the UI.graphics and UI.menu are shown Figure 31. For example, when a mouse click is detected, the event is broadcast to all the visual objects (components, ports, couplings). Except for the couplings, at most one visual object should accept the mouse event and pop up the associated menu as described in the Section 7.1.3. For couplings all the overlaid couplings are displayed in the menu (see Figure 34).

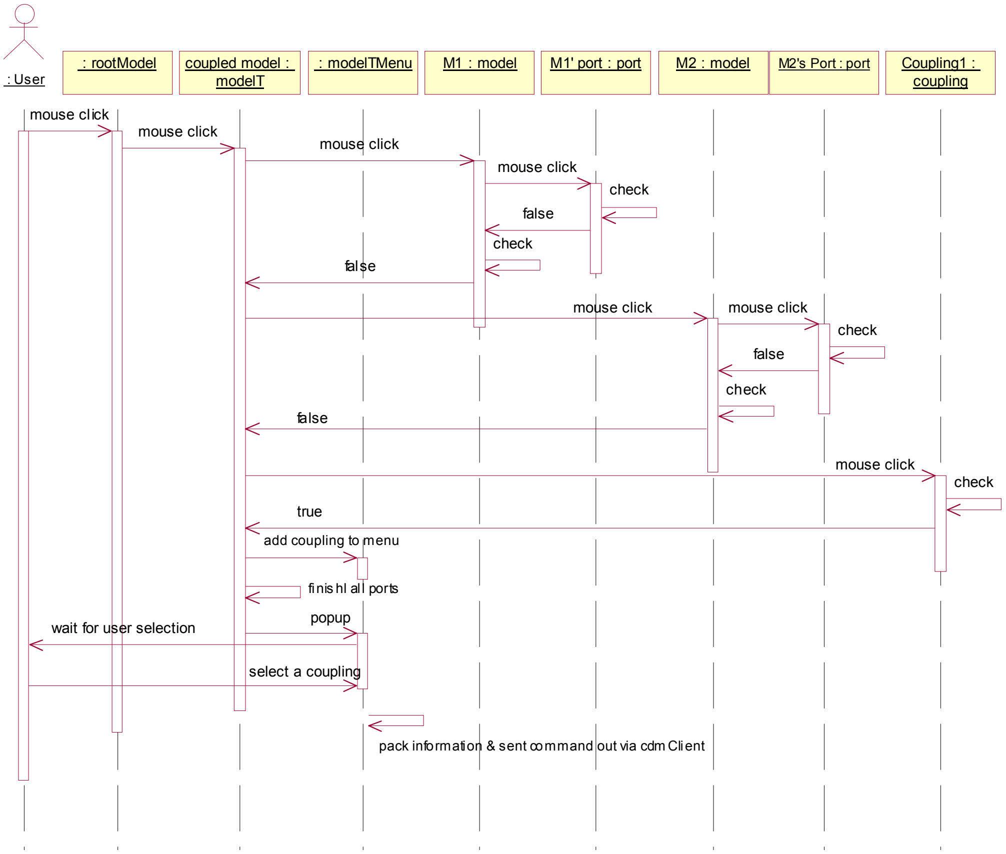


Figure 31: Graphics and Menu Interaction Diagram

7.3.3. UI.menu package

The UI.menu package offers pop-up menus allowing users to manipulate the model visually. As described earlier, each menu object is tailored for a specific type of visual objects. Transactions that are necessary for manipulating models are displayed to a user as menus based on the type of visual object. For example, the menu for a model component displays five choices: Delete, Rename, Add In Port, Add Out Port, and Add Component (see Figure 33). For couplings, the user can view the set of couplings between two components and have the choice of deleting them (see Figure 34). The menu obtains model data from the visual object and required inputs from the user. The relationship between visual objects and menus are shown in the Figure 32.

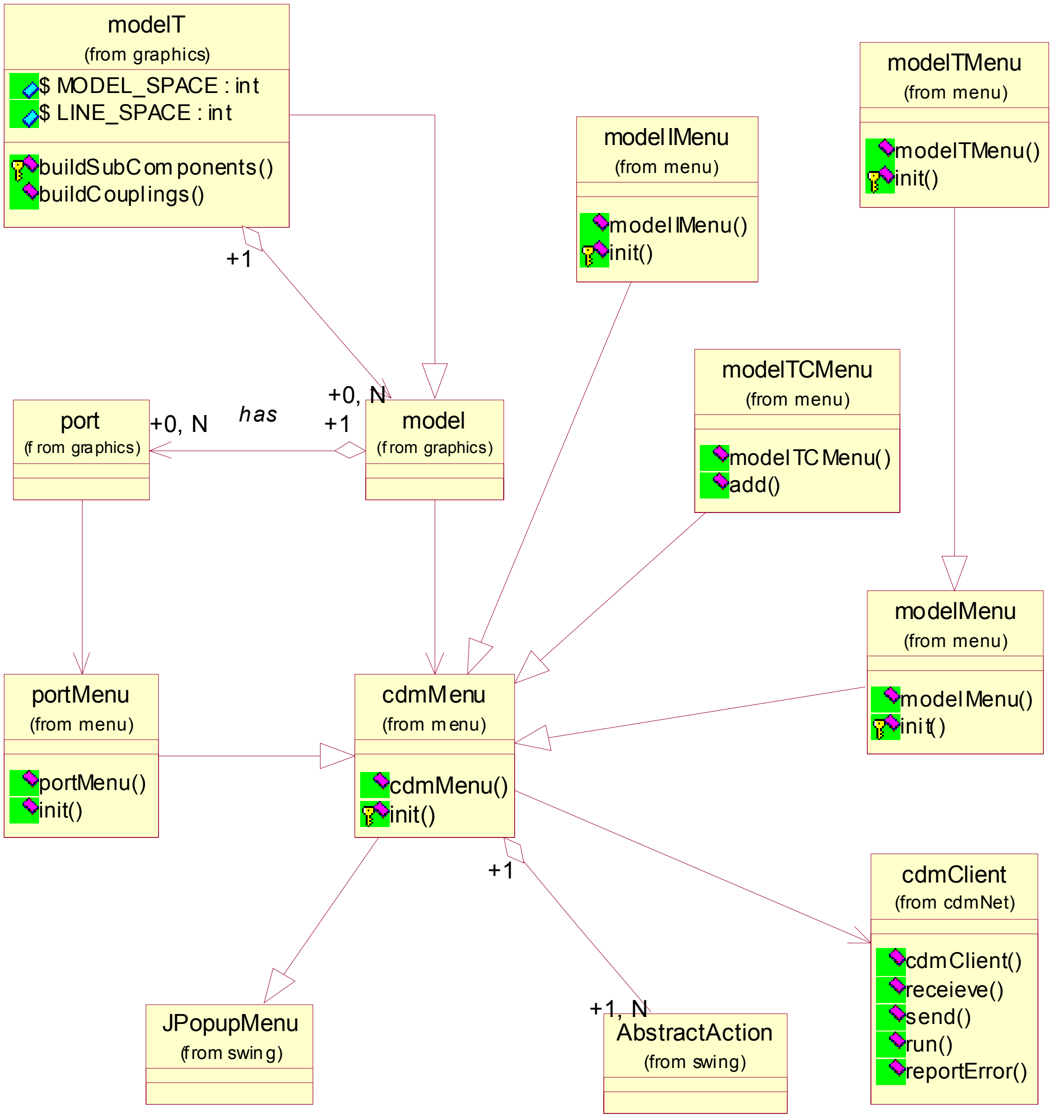


Figure 32: UI.menu Class Diagram

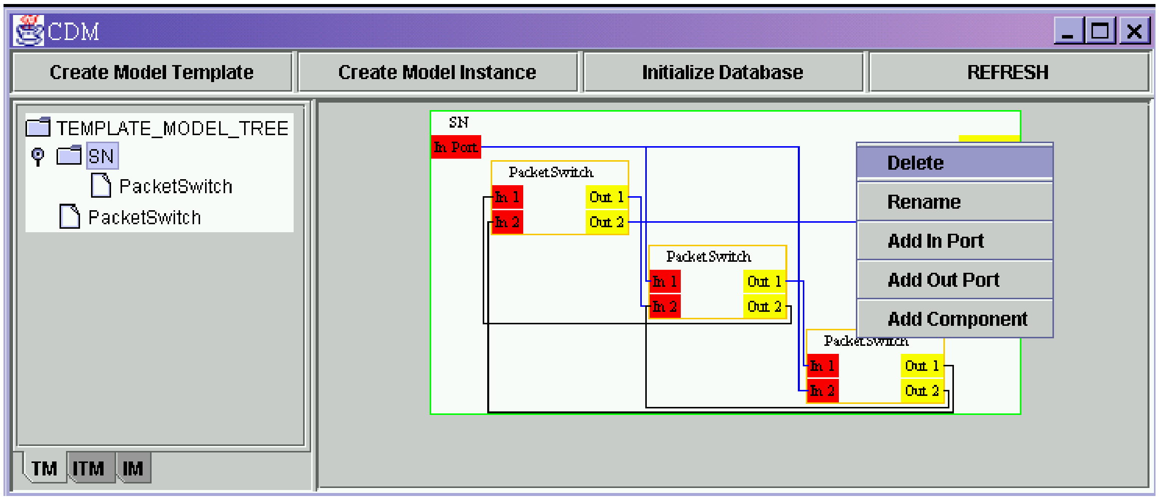


Figure 33: Popup Menu Screen Shot

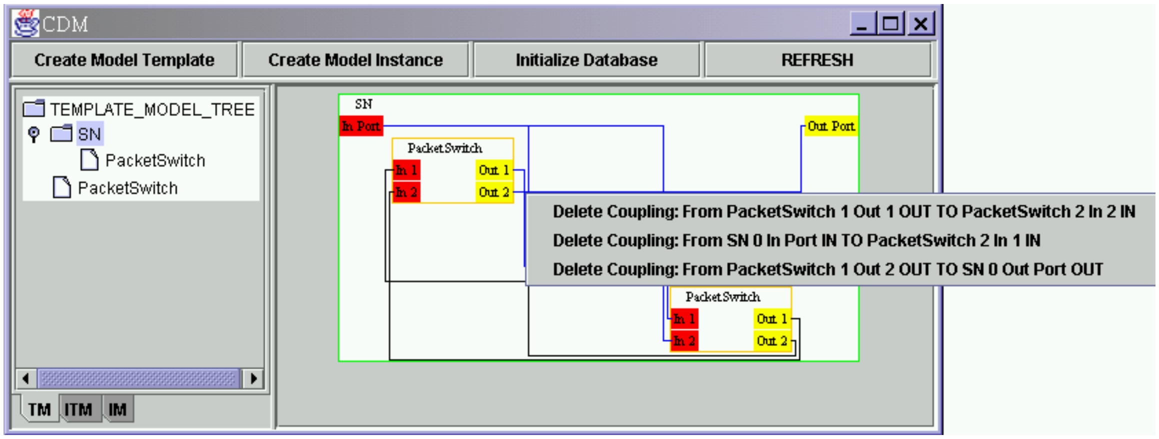

CDM
Create Model Template
Create Model Instance
Initialize Database
REFRESH
TEMPLATE_MODEL_TREE
SN
PacketSwitch
PacketSwitch
SN
In Port
Out Port
PacketSwitch
In 1
In 2
Out 1
Out 2
Delete Coupling: From PacketSwitch 1 Out 1 OUT TO PacketSwitch 2 In 2 IN
Delete Coupling: From SN 0 In Port IN TO PacketSwitch 2 In 1 IN
Delete Coupling: From PacketSwitch 1 Out 2 OUT TO SN 0 Out Port OUT
PacketSwitch
In 1
In 2
Out 1
Out 2
TM
ITM
IM

Figure 34: Coupling Menu Screen

8. SESM PROTOTYPE IMPLEMENTATION

8.1. Java Database Connectivity

8.1.1. Overview of JDBC

The JDBC API provides database connectivity [Cat 97]. Using JDBC, an application can access a database independent of the actual database management system being used for data storage. A Java application using JDBC can access any one of three major database architectures – relational, object, or object-relational databases. However, the JDBC API is heavily biased towards relational database and its standard query language (SQL) because of the overwhelming usage of relational databases [Ree 00]. Mainly, JDBC is a SQL-level API that allows Java programmers to embed SQL statements as arguments in the methods provided by JDBC interfaces. Most major database vendors provide run-time implementation of the JDBC interfaces, generally referred to as JDBC drivers. The implementation of a JDBC driver will route the SQL statements in some proprietary fashion a given database management system can recognize.

8.1.2. Four Types of JDBC

Currently, there are four different types of JDBC drivers. These drivers are (I) JDBC-ODBC Bridge, (II) Native-API Partly Java, (III) Net-protocol All-Java, and (IV) Native-protocol All-Java Drivers [Ree 00]. A Type I driver uses a bridge technology to connect a Java client to an ODBC database system. This type of driver requires non-Java software to be installed on the client, and the drivers are implemented using native code. The type II driver uses a native code library to access a database, wrapping a thin layer of Java around the native library. For instance, with Oracle databases, the native access is through the Oracle Call Interface (OCI) libraries that were originally designed for C/C++ programmers [Ree 00]. Type II drivers are also implemented with native code. Use of Type I and II drivers poses some element of risk since a defect in the native code will crash the Java Virtual Machine. Type III drivers define a generic network protocol that interfaces with a piece of custom middleware. The middleware component provides the actual database access [Ree 00]. Type IV drivers are written entirely in Java [Ree 00]. The clients who incorporate these drivers can access the database directly without any additional software. However, type IV drivers require that the Java security manager allow TCP/IP connections to the database server. Both Type III and IV drivers are especially useful for applet deployment, since the actual JDBC classes can be written entirely in Java and downloaded by the client at run-time.

8.1.3. Objects and JDBC

Although JDBC provides access to a relational database, the object-to-relational mapping problem is left for the application developer [Cat 97]. As mentioned earlier, a relational database is designed to contain relational data elements. However, an object-oriented language like Java, requires that an object's characterization to contain data with methods operating upon them together. As a result, the application developer must create a translation layer that transforms an object's data into relations (generally a set of interrelated tables) and conversely provides a way to relate data in a relational database to data contained in objects. For instance, a SESM model is treated as an object in Java. Yet, the information regarding the model is stored in several tables. Therefore, the translation layer should create the relational representation of the SESM model object and provide methods to retrieve required data from the relational database.

8.2. Oracle8i DBMS

8.2.1. Family of Oracle 8i DBMS

The Oracle8i product line includes Oracle8i, Oracle8i Enterprise Edition (EE), and Oracle8i Personal Edition (PE) [Ora 02]. The Oracle8i has easy to use features like integrated management tools, full distribution, replication, and web enabled. The Oracle8i PE supports single user development and deployment that require full compatibility with oracle8i and Oracle8i EE. The Oracle8i EE supports enterprise level applications (distributed access and scalability) with additional tools and functionality. These Oracle8i products are all built using the same database engine architecture [Ora 02]. Thus, Both Oracle8i and Oracle8i Personal Edition are 100 percent compatible with Oracle8i Enterprise Edition on many platforms. As a result, users can select appropriate Oracle8i product depending on their platform and use the same application without re-engineering.

8.2.2. Integrated management Tools

Other than compatible products for different needs and platforms, Oracle8i also provide integrated management tools. The Oracle Enterprise Manager is a 3-tier management tool for Oracle8i [Ora 02]. The Oracle Enterprise Manager console (the GUI of the Manager) is a Java based application that can be executed as installed Java application or from a Java enabled browser. Furthermore, the Java based applications, Oracle Universal Installer and Oracle Database Configuration Assistant, are included in Oracle8i products. Utility tools such as install, pre-tune, and configure provide automatic configuration by detecting hardware characteristic and user's preferences. These utility tools have improved the usability of and reduce Oracle8i's management cost.

8.2.3. Extended Java Support

Finally, the Oracle8i provides extended support for Java. For instance, The Oracle8*i* Java Virtual Machine (JVM) constitutes the heart of Oracle8i's support for Java [Ora 02]. The Oracle8i JVM is a server side Java engine for the Oracle8i database. It is a Java Virtual Machine with a native compiler, a CORBA 2.0 ORB, an EJB server, an embedded server side JDBC driver, and a SQLJ translator [Ora 02]. Although the Oracle8*i* JVM is developed by Oracle, the JVM is 100% JDK compliant. Furthermore, Oracle8i offers three different types of JDBC driver, JDBC Thin Client Side, JDBC OCI Client-Side, and JDBC Server [Ora 02]. The Oracle JDBC Thin driver is a Type IV driver that is written in 100% pure Java and complies with the JDBC 1.22 standard. For communication with the database, the driver includes an equivalent implementation of Oracle's Two-Task Common (TTC) presentation protocol and Net8 session protocol in Java. Both of these protocols are lightweight implementation versions of their counterparts on the server [Ora 02]. The Net8 protocol runs over TCP/IP only. To use this driver, it is not necessary to install any Oracle-specific software on the client.

The JDBC OCI driver is a Type II driver that is targeted to client-server Java applications and Java-based middle-tier [Ora 02]. As mentioned earlier, the JDBC OCI driver converts JDBC invocations to calls to the Oracle Call Interface (OCI). The JDBC OCI driver is written in a combination of Java and C. The driver requires the OCI libraries, Net8, CORE libraries, and other necessary files on each client to be installed on each client. The JDBC Server driver allows Java programs that use the Oracle 8.1.5 Java Virtual Machine (VM) and run inside the database to communicate with the SQL engine. The Server driver, the Java VM, the database, the KPRB (server-side) C library, and the SQL engine are all executed within the same address space. There are no network round-trips involved. The programs access the SQL engine by using function calls.

8.3. Implementation

The SESM prototype is a single machine, single user system implemented using Java. One of the reasons for choosing Java as the programming language is that Java provides the Java Database Connectivity (JDBC) API. The Oracle8i DBMS Enterprise Edition on Windows 2000 was selected as the main relational database system platform. One of the main reasons of choosing Oracle8i was its flexibility, usability, and support for Java. The SESM server utilizes the JDBC Thin Client Side driver to connect to the Oracle8i. The JDBC Thin Driver is portable and does not require additional installation. Both the SESM Server and Client are implemented as designed (see Chapter 6 & 7). The client and server are executed on separate threads and communicate with each other via the Network Environment. The Network Environment is implemented using shared memory.

9. CONCLUSIONS

In this thesis, we described our approach to the development of the Scaleable System Entity Structure Modeler (SESM) environment. To support modeling of large-scale systems, we devised an entity relationship diagram for capturing the necessary modeling relationships (e.g., decomposition and specialization among model components). The tool offers modeling constructs based on the key System Entity Structure concepts and those that underlie capturing template, instance template, and instance models. Furthermore, the entity relationship captures input/output ports and couplings for model components. Next, we discuss our two key objectives: support modeling of large-scale systems and ease of modeling via a friendly user interface.

The SESM's software architecture (composed of Server, Client, RDBMS and Network Environment) is primarily client-server type. The client-server architecture offers simplicity and in particular supports higher degree of modular design and implementation. One of the most important parts of the architecture is RDBMS. Our choice of relational database enables modelers to construct and store hierarchical models based on proven and scaleable features of relational database management systems. Use of relational database management system provides primitive support for distributed clients. That is, SESM can use multiple read and write operations that are guaranteed to be correct based on native (automatic) synchronization and locking mechanisms offered by RDBMS. Both the Server and Client interact with the RDBMS via the Network Environment. To increase performance, we proposed direct and indirect (i.e., via the Network Environment and Server) read operations. Direct read operations results in superior performance since generally there are may more read operations than write operations.

The representation of models in a relational database required mapping of the object-oriented models (e.g., DEVS coupled models) into their relational counterparts (i.e., a set of interrelated tables). Two general types of operations (i.e., modification and query) were provided by SESM. The selection of relational database also required model operations (e.g., adding a component) to be mapped into long-transaction in SQL and programming logic. The Server and Client programming logic is used to account for modeling constrains that cannot be directly expressed using relational model and relational algebra.

The Graphical User Interface of SESM plays a major role in supporting model development. The visualization window offers complementary views of models (tree structure and block-diagram with ports and couplings). Indeed, a hierarchical model composed of several hundred components, ports, and couplings require friendly visualization. Through SESM's user interface, end-users (e.g., subject matter experts) can develop models in an orderly fashion. In particular, given the three types of models (template, instance template, and instance models), it was necessary to design a User Interface to help a user start with template model and then follow with instance and instance model creations.

We have considered and studied some additional important features for the SESM environment. We discuss these in the next section. Aside from these, it is also important for SESM User Interface to enable users to create multiple models instead of having a single model containing many different sub-models. An important benefit of this feature is that modelers can save intermediate models separately and therefore have the option of going back to earlier versions if there is a need. RDBMS, however, does not provide direct support for this feature. For example, in Oracle databases, all models developed in SESM environment are stored in one or more Table Spaces that share the same schema. In contrast, using MS Access, each model can have its own separate database file.

9.1. Future Works

9.1.1. Support for Dynamic Modeling

The kinds of modeling supported by SESM are primarily structural – i.e., a model is represented in terms of its input/output interface and how increasingly complex models can be constructed hierarchically. It is important to specify how a component can process inputs and generate outputs. For example, it is very important to be able to represent the dynamics of DEVS atomic models (state variables, internal, external, and output transition functions). With SESM supporting modeling of atomic model supported, it can be used to not only model DEVS models, but also used with simulation engines. Therefore, we can foresee a collaborative modeling environment tied to a distributed simulation engine which can support large-scale modeling and simulation using the DEVS modeling and simulation framework.

9.1.2. Support for Collaborative Model Development

The SESM environment is developed primarily for a single user. However, as mentioned above, RDBMS support for multiple users is inadequate as compared, for example, with the Collaborative DEVS modeler. Higher-level capabilities such as keeping track of the modelers' identity, joining/leaving a modeling session, and coordination of clients' activities (e.g., ensuring correct sequencing of events such as adding a component, deleting a coupling, etc.) are not supported by SESM. This suggests a new architecture design that uses a suitable middleware (e.g., Collaborative Distributed Network System) instead of the Network Environment.

9.1.3. Advanced Visualization

There are several improvements for the visualization that can be considered for future developments. First, the method of placing component models diagonally does not make best use of available space. Achieving compact block diagram representations while providing best readability is an important consideration. Having the ability to draw block diagrams compactly results in a view that can contain several levels of a hierarchical model. Second, crossings among couplings often make identifying source and destination of couplings difficult – e.g., it would be difficult to visually distinguish couplings (multiple output ports of one component connected to multiple input ports of another component) without the use of pop-up screens that display coupling lists. However, it is well known that minimizing crossings is an NP hard problem. Advances in this area combined with SESM domain-specific modeling requirements (types of couplings) may lead to heuristics approaches for reducing crossings. For example, we can group multiple couplings between two components into one super coupling using fan-out and fan-in constructs of input and output ports – i.e., group a component's multiple output ports into one super output port and multiple input ports into one super input port, respectively. This approach, however, does not show a one-to-one mapping between output and input port couplings. A complementary strategy is to identify ports that have only one coupling and then reorder the ports (and couplings) to reduce crossings.

REFERENCES:

[Bla 98] Blaha, M. and Premerlani, W., *Object-Oriented Modeling And Design For Database Application*, Upper Saddle River, N.J.: Prentice Hall, c1998.

[Boo 94] Grady B., *Object-Oriented Analysis and Design with Applications*, 2nd Edition, Cunning Benjamin, 1994

[Bar 93] Bartels, D. "ODMG 93-The Emerging Object Database Standard," *Proceedings of the Twelfth International Conference on Data Engineering*, IEEE Comput. Los Alamitos, CA: Soc. Press, 1996

[Cat 97] Cattell, R.G.G. and Fisher, Maydene, *JDBC Database Access with Java*, Reading, Mass: Addison-Wesley, 1997.

[Che 76] Chen, P., "The Entity Relationship Model – Toward A Unified View Of Data," *ACM-Transactions-on-Database-Systems*. Vol.1, No.1, p.9-36, March 1976

[Elm 94] Elmasri, R. A., Navathe, S.B., *Fundamentals of Database Systems*, 2nd Edition. Addison Wesley Publishing Company. 1994.

[Fu 01] Fu, T., Hierarchical Modeling Using a Graphical User Interface, Independent Study Report, unpublished report, Electrical and Computer Engineering Dept., University of Arizona, July 2001.

[For 99] Fortier, P. J, *SQL 3: Implementing The Object-Relational Database*, New York: McGraw-Hill, 1999.

[Fow 99] Fowler, M. and Scott, K., *UML Distilled: A Brief Guide to the Standard Object Modeling Language,* 2nd Edition, Addison-Wesley Pub Co, August 20, 1999

[Kam 91] Kamada, T. and Kawai S., "A General Framework for Visualizing Abstract Objects and Relations," *ACM Transactions on Graphics*, Vol. 10. No. 1, January 1991

[Lon 00] Loney, K. and Koch, G., *Oracle8i: The Complete Reference*, 1st Edition, McGraw-Hill Professional Publishing, January 15, 2000

[Mul 99] Muller, R.J., *Database Design For Smarties: Using UML For Data Modeling*, San Francisco: Morgan Kaufmann Publishers, c1999.

[Nai 01] Naiburg, E., *UML for Database Design*, Boston: Addison-Wesley, c2001

[O'Ne 01] O'Neil, Patrick, *Database--Principles, Programming, and Performance*, 2nd Ed., San Francisco: Morgan Kaufmann Publishers, c2001.

[Ora 02] http://technet.oracle.com, 2002.

[Pap 00] Papazoglou, M.P., Spaccapietra, Stefano, and Tari, Zahir, *Advances in Object-Oriented Data Modeling*, Cambridge, Mass.: MIT Press, 2000.

[Par 94] Park, H.-C., Lee, W.-B., Kim, T.G., "A Relational Algebraic Framework For Models Management," *1994 Winter Simulation Conference Proceedings,* New York, NY, USA: IEEE, p.649-656, 1994

[Par 96] Park, H.C., Kim, T.G., "Relational Algebraic System Entity Structure For Models Management", *IEEE Proceedings Computers and Digital Techniques*. vol.143, no.1, p.49-54, Jan. 1996.

[Par 97] Park, H.-C., Lee, W.-B, Kim, T.G., "RASES: A Database Supported Framework For Structured Model Base Management," *Simulation Practice and Theory*. vol.5, no.4, p.289-313, 15 May 1997.

[Par 98] Park, S., Collaborative Distributed Network System Architecture: Design and Implementation, Electrical and Computer Engineering, Masters Thesis, University of Arizona, 1998.

[Ram 00] Rumbaugh, J., Jacobson, I., and Booch, G., *The Unified Modeling Language Reference Manual*, Addison Wesley

[Ree 00] Reese, G., *Database Programming with JDBC and Java,* 2nd Edition, O'Reilly & Associates, January 15, 2000

[Roz 83] Rozenblit, J.W. and Zeigler, B.P., "Representing and Construction System Specifications Using the System Entity Structure Concepts", *Proceedings of the 1993 Winter Simulation Conference*, p. 604-611, Los Angeles, December 1993.

[Roz 89] Rozenblit, J.W., Hu, J.F. and Huang, Y.M., "An Integrated, Entity-Based Knowledge Representation Scheme for System Design", *Proceedings of the 1989 National Science Foundation Design Research Conference*, p. 393-408, Amherst, Mass, June 1989.

[Sar 99] Sarjoughian, H.S., Nutaro, J.J., Zeigler, B.P., "Collaborative DEVS Modeler, Western Multi-Conference on Web-based Modeling and Simulation", *SCS*, 1999.

[Sar 00] Sarjoughian, H.S, Park, S., Zeigler, B.P., "Collaborative Distributed Network System: A Lightweight Middleware Supporting Collaborative DEVS Modeling", *FGPC*, Vol 17, p. 89-105,2001.

[Sar 02] Sarjoughian, H.S, Fu, Ting-Shen, "An Approach for Scaleable Model Representation and Management Methodology", in preparation, 2002.

[Sto 96] Stonebraker, M. *Object-Relational DBMSs: The Next Great Wave*, San Francisco, Calif.: Morgan Kaufmann Publishers, c1996.

[Sun 00] Sunderraman, Rajshekhar, *Oracle8 Programming: A Primer*. Addison Wesley 2000

[Sug 91] Sugiyama, K., Misue, K., "Visualization of Structural Information: Automatic Drawing of Compound Digraphs," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 21 No. 4, July/August 1991

[Zei 84] Zeigler, B.P., *Multifacetted Modelling and Discrete Event Simulation*, London: Academic Press, 1984.

[Zei 90] Zeigler, B.P., Object Oriented Simulation With Hierarchical Modular Models: Intelligent Agents and Endomorphic Systems, Academic Press, 1990.

[Zei 00] Zeigler, B.P., Praehofer, H., Kim, T.G., Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems, 2nd Edition, Academic Press, 2000.