Chapter 7

Data Models For Metrics-Based Project Management Systems

7.1 Introduction

Existing work in software metrics has focused mainly on the type of metrics to be collected and the analysis of metrics data, particularly with respect to complexity and reliability analysis of large software programs. The data model of the underlying database management system in which the metrics data is stored has not been given a serious consideration by software researchers to date, since they usually adopt one of various data models to store their software metrics data with little regard to their relative efficiency. We believe, however, that the underlying data model should not be taken for granted. The adoption of an appropriate data model, geared to the nature of project-related queries, greatly improves the efficiency of queries and the maintainability of a software metrics management system, if we can predict in advance the type of queries that will most likely be posed. The nature of queries can vary from user to user. In this chapter we focus our attention on developing a data model suited towards the management of software metrics databases.

We begin our discussion by first describing the principles and operation of various existing data models, and then analyze the suitability of each data model with respect to various queries for the software metrics management system. In particular, we focus on the data modeling aspects of software metrics and discus issues related to *user-independent view* and semantic heterogeneity pertaining to quality and risk management. A general purpose software metrics management system should provide an environment for users to express and for the system to process semantically heterogeneous queries. Toward this goal, we propose an object-oriented temporal modeling framework for software metrics data that allows users to capture

temporal semantics of information associated with software to develop a semantically-rich information space.

The organization of this chapter is as follows. We begin this chapter (Sections 7.2 - 7.4) with a detailed discussion on data modeling approaches and elaborate pros and cons of various approaches. We then in Section 7.5 present a formal approach for semantic modeling of metrics data in terms of supporting the view of quality and risk. Towards the goal, we introduce a Petri Net model for capturing temporal relationships among metrics data and outline a methodology for formally characterizing "scenarios" embedded in metrics data. For this purpose, *generalized n-ary temporal relations* are introduced in this section. An object-oriented paradigm is proposed in Section 7.5 to categorize scenarios as classes and to provide a powerful abstraction mechanism for the management team for measuring quality and risk of software projects. In Section 7.6, we provide performance comparison between relational and graph-based database management systems in terms of their suitability to support object-oriented database applications.

7.2 Overview of Data Models for Software Metrics Database Management

Many data models have been proposed in the literature [CHE76, GRA95, GUN91, KUN90, KUP93, LEV91], and can be categorized based on the types of concepts they provide to describe the database structure. High-level or conceptual data models provide concepts that are close to the way many users perceive data, whereas low-level or physical data models provide concepts that describe the details of how data is stored in the computer. Examples of high-level data models include the relational data model and the entity-relationship (E-R) data model discussed in a previous chapter. We begin on discussing by exploring the efficiency of various data models with respect to performing various types of queries. In particular, we analyze the efficiency of the popular relational model with respect to both simple queries as well as more complex queries involving temporal data, as well as recursive queries. We then explore the

effectiveness of alternative models such as the graph data model and object-oriented data model with respect to the same set of queries.

7. 2.1 METRICS QUERIES USING THE RELATIONAL DATA MODEL

The relational model was defined by Codd [COD70], and has been used by many database management systems including initial research prototypes such as System R [AST76,CHA81] and INGRES [STO76, STO80], as well as popular commercial products like Oracle, Sybase, Informix, and Access. We now describe various kinds of queries on metrics databases, and examine the suitability of the relational model for supporting them. Many straightforward queries to metric databases are natural under the relational model.

7.2.1.1 Simple Selection Queries

Simple and straightforward queries can easily be accommodated by the relational model. A schedule-related query that a manager may ask in a software engineering project may be*: What is the planned start date for the Preliminary Design milestone?* . Suppose we have a *SCHEDULE* relation with the tuples *<SYSTEM, ACTIVITY, PLANNED-START-DATE, PLANNED-END-DATE, ACTUAL-START-DATE, ACTUAL-END-DATE>*. We can obtain the required information using simple SQL queries on the relational database as illustrated in Figure 7.1.

```
SELECT PLANNED-START-DATE
FROM SCHEDULE
WHERE ACTIVITY="PRELIMINARY-DESIGN"
```

Figure 7.1: Query on Planned Start Date Using the Relational Model

7.2.1.2 Temporal Queries

Many project-related queries are temporal in nature. The reason is that managers would like to plot graphs to view current project trends and predict future project directions. For

example, a variation of the schedule-related query discussed in the previous section could be: *What is the total slippage in the planned start date for the Preliminary Design milestone?* Slippage in the planned start date is the difference between the planned start date and actual start date for a given milestone. Therefore, if the most recent planned start date for the milestone is Project Month 2 and the actual start date for the milestone is Project Month 3, then the slippage in the planned start date is one month. The total slippage in the planned start date for a given milestone is the difference between the planned start dates between the current project month and the start of the project.

Note that the relational model updates data in place. Since no temporal data is stored in the given SCHEDULE relation, the planned and actual start dates get overwritten at the end of every month, or whenever schedules are modified by management. An additional field can be added to the relational database to store the temporal data, by extending the SCHEDULE relation to include a DATA-DATE attribute, so that it now becomes *<DATA-DATE, SYSTEM, ACTIVITY, PLANNED-START-DATE, PLANNED-END-DATE, ACTUAL-START-DATE, ACTUAL-END-DATE>*. This query can then be answered indirectly from analysis of the *PLANNED-START-DATE* attribute between the original planned project start dates and the current planned project start dates. Assuming that the project started before January 31, 1995 and that the most recent data was entered after October 31, 1995, the cumulative slip start date can be answered using a query as illustrated in Figure 7.2.

```
SELECT PLANNED-START-DATE
FROM SCHEDULE
WHERE ACTIVITY="PRELIMINARY-DESIGN" AND
      DATA-DATE<="31-JAN-95" or DATA-DATE>="31-OCT-95".
```

Figure 7.2: Query On Schedule Slippage Using Relational Model

The above query may result in multiple occurrences of the result depending on the frequency with which the planned start date has been modified, and more effort is needed to retrieve the result with the earliest DATA-DATE field.

```
SELECT COVERAGE, TEST-SUCCESS, OVERALL-SUCCESS
FROM BREADTH-OF-TESTING
WHERE MODULE-NAME in
        ("XYZ" OR
                SELECT CONSTITUENT-MODULE
                FROM MODULE
                WHERE PARENT-MODULE="XYZ"
        OR      SELECT CONSTITUENT-MODULE
                FROM MODULE
                WHERE PARENT-MODULE=
                        (SELECT CONSTITUENT-MODULE
                        FROM MODULE
                        WHERE PARENT-MODULE="XYZ")
        OR      SELECT CONSTITUENT-MODULE
                FROM MODULE
                WHERE PARENT-MODULE=
                        (SELECT CONSTITUENT-MODULE
                        FROM MODULE
                        WHERE PARENT-MODULE=
                                SELECT CONSTITUENT-MODULE
                                FROM MODULE
                                WHERE PARENT-MODULE="XYZ")
        OR .....        and so on until the maximum nesting level of modules, if
defined
        )
```

Figure 7.3: Attempt at Recursive Query Using the Relational Model

7.2.1.3 Recursive Queries

A manager may also pose queries to the software database management related to the reliability of the project, such as the following: *How thoroughly has the software project XYZ been tested?* This involves both the *Breadth of Testing* and *Depth of Testing* [ARM92] metrics. The *Breadth of Testing* metric addresses the degree to which required functionality has been

successfully demonstrated. This can be called *black box* testing. The *Depth of Testing* metric provides indications of the extent and success of testing from the point of view of coverage of possible paths and conditions within the software. Since a software project consists of a number of modules, which may themselves consist of further sub-modules, the response to this query requires determining both *Breadth of Testing* and *Depth of Testing* of not only the overall software project itself, but also of each of the modules that comprise the software project, that is, the query is recursive. Assume we have a *BREADTH-OF-TESTING* relation to store metrics data for the *Breadth of Testing* metric, consisting of the tuples *<DATA-DATE, MODULE-NAME, COVERAGE, TEST-SUCCESS, OVERALL-SUCCESS>* and we also have a separate relation *MODULE* to store the names of modules that make up a given module with the tuples *<PARENT-MODULE, CONSTITUENT-MODULE>*. Making the (often) unrealistic assumption that there is a maximum nesting to the number of levels of submodules, the result is a complicated query, as illustrated in Figure 7.3.

7.2.1.4 Strengths of the Relational Model for Metric Databases

The relational model is simple to understand and use for straightforward queries. It is suitable for a large number of metrics oriented queries since many queries are straightforward and only require simple retrieval and analysis of data. Furthermore, since many users in the data management community are familiar with relational technology, relational databases, though inadequate for more complex queries, have gained widespread support and are used in many software metrics applications.

7.2.1.5 Weakness of the Relational Model for Metrics Databases

The relational model has several shortcomings with respect to developing a general purpose software metrics database. The model causes *segmentation*, as it cannot properly decompose logically coherent application objects over several base relations. For example,

assume we need to store information relating to software packages in the metrics. A software package is comprised of modules. A module forms one composite object that should really be viewed as one coherent unit, with attributes such as lines of code, complexity rating, and percentage of tested paths associated with the object. These attributes are associated with the object as a unit, not with the partitions thereof that may be stored in the various relations, depending on how the database is structured. Likewise, we may want to associate operations with module objects, such as finding the number of paths for each module, etc.

The relational model suffers from a lack of abstraction. It has only one structuring concept, the relation. This is sufficient for data processing applications where objects exhibit a very homogeneous structure, but not in engineering applications where objects possess a more heterogeneous structure. A complex object, such as a module in a software metric application, may be composed of a variety of differently structured subobjects. In the relational model, we must represent a module by splitting its features into different relations; for example, with one relation containing the list of submodules that comprise the module, and another relation for details on the number of data paths and complexity rating for that module.

As we can see from the schedule scenarios, the relational model does not address the temporal dimension of data. This is regarded as a severe drawback for a software metric management system which are regularly updated but which require past, present, and future data values to be dealt with by the database. We have partially worked around this problem in our example scenario by including a "date" field in the tuples so as not to overwrite existing data; however, this does not allow, for example, for entries to be "sorted" by date since relational databases do not order data. It is possible to order tuples retrieved in a query by the date field by using the *ORDERBY* clause of SQL. Similarly, one could order the tuples in the physical storage by the date field by explicitly specifying this to the physical storage mechanism. However, in

either case there is the extra burden placed on the user since the system does not understand the concept of time and its basic properties like ordering and causality.

The inability to perform *recursion* in the relational model is highly restrictive in complex applications. A good example is given in the reliability query discussed in this section relating to how thoroughly a software project has been tested.

7.2.1.6 Extensions to the Relational Model

To overcome the restrictions imposed by the relational model, a number of extensions to relational DBMS have been proposed, to model generalization and aggregation, abstract data types, temporal properties, recursion, etc. Recent efforts, most notably those in the draft standard SQL3, aim at adding the object-oriented concepts of generalization and aggregation, and abstract data types in SQL. However, given the complex and unwieldy nature of SQL3, and other cleaner proposed object data model standards such as ODMG93 [CAT94], it is not clear that extending the relational model is the right approach to achieving object capabilities in a data model.

Research in temporal databases [OZS95] has focused on enhancing the relational model with built in support for the temporal dimension. A number of models have been proposed where the goal has been to make the system understand special properties of the time dimension like ordering of time points, temporal precedence associated with causality, temporal proximity, etc. The TSQL2 standard [SNO94] is a consolidation of more than a decade of work in this area, and is aimed at extending the SQL standard with temporal semantics.

Motivated by the integration of databases and logic programming, the field of deductive databases [CER89, MIN88] has contributed a lot our understanding of extending the computational power of database programming languages [CHA88]. Special attention has been given to extending the power of relational query languages by adding recursion, a capability that has been sorely missed for a long time. Dar [DAR93] extended SQL to form SQL/TC, to allow

the expression of generalized transitive closure queries, in order to permit the user to pose queries that compute paths between two points and information associated with these paths. Such queries may specify selections on arcs, paths, or set of paths. The output of a query may include the aggregation of information for different paths between the same endpoints. Although SQL does not directly support recursive search operations such as transitive closure or fix-point, Kamfonas [KAM92] established a design approach whereby transitive closure search operations especially on tree structures like ancestor-descendent queries can be performed naturally in SQL. Eder [EDE90] also proposed an extension to SQL for the processing of recursive structures and solving general transitive closure problems, and integrated this with the view definition mechanism of SQL. This construct is based on a generalization of transitive closure and is formally defined. Because of the importance of extreme value selections, special constructs are introduced for the selection of tuples with minimal or maximal values in some attributes. Applying these selections on recursively defined views constitutes nonlinear recursion, necessitating the introduction of new special constructs. A lot of work has been done in the general area of extending SQL to support recursion, and a good survey is provided in [BAN86].

7.3 Metrics Queries Using The Object-Oriented Data Model

The research community has already embraced the object-oriented data model. A principal difference between a traditional relational (or network or hierarchical) DBMS and an object-oriented DBMS is in the passive and active behavior of the underlying system, and the manner in which these are implemented. In an object-oriented database, the database contains objects that are made up of both passive data and active data to reflect the behavior of the object. For example, if the object is a module, then besides data structured to represent the module such as the number of lines of source code and number of decision paths, and the actual source code for the module, it contains the code that represents the behavior of the module, such as the

calculation of complexity ratings, the thoroughness of testing, etc. In a software metric application, for example, a module object may call upon other module objects to calculate the thoroughness of testing, and the latter may call upon yet other module objects, to satisfy the recursive request. Another principal feature of object data models is the support for generalization and specialization hierarchies. This enables a user to easily model a system which has a complex structure of modules and submodules. Some object-oriented data models, especially those for design and engineering applications, provide built-in features for supporting multiple versions of data. This can allow the modeling of temporal data in a natural way. For example, in a temporal database, different versions of a schedule object may store project schedules for a given milestone for different time periods.

7.3.1 Formulation of Metric Database Queries

The object-oriented model allows natural representation of many of the entities in a software configuration. For example, software modules can be modeled as objects using the O-O model, with attributes such as the number of lines of source code, complexity rating, number of decision paths, and so on. The entire software project can also be modeled as a single complex object, encapsulating attributes such as cost, schedule and so on. New attributes can therefore be added or the structure of existing attributes modified without any change required to the application program posing the query to the software metrics management system.

We now express the partial schema for our metrics query example for our hypothetical database in the object-oriented data model in Figure 7.4, using C++ as the data definition language. The implementation of the **Mile_Get_Latest_Slip_Start_Date** method for a given milestone M could look like this in pseudo-code:

1. Look up the container class **M.MileSchedule** in the **ooMileStone** class for schedule with the most recent and second most recent planned start dates. This is given in the **SchedPlanned_Start_Date** field.
2. If there is any difference in the **SchedPlanned_Start_Date** fields, return the difference, else return 0.

The other queries relating to planned end dates are handled similarly. The application therefore simply needs to invoke the method for milestone M by

M. Mile_Get_Latest_Slip_Start_Date ( )

to obtain a solution to the query.

```
// This is the header file to be included for all object-oriented database applications
#include <oodefs.h>

class ooMetricsDB;
class ooMilestone;
class ooBreadthofTesting;
class ooSchedule;
class ooModule;

// ... and other classes ...
// this is the MPMS object for a single project. This object would contain, amongst other information, a list of milestones
// each with associated schedules, and a list of modules.
class ooMetricsDB : public ooDBClass {
        ooDBContainer(ooMetMileStone) MetMilestone;
        ooDBContainer(ooModule) MetModule;
};
//  The milestone object contains the name of the milestone and the schedule for that milestone, etc.
class ooMilestone : public ooDBClass {
        ooString MileName;
        ooContainer(ooSchedule) MileSchedule;
//  ... and other definitions
//  These are methods for the ooMilestone object. The methods illustrated here returns the latest slip start and end dates
//  for the milestone, and the cumulative slip start and end dates for the milestone
public:
```

```
// The methods below return an ooInteger representing the number of months the schedule slipped.
//
        ooInteger Mile_Get_Latest_Slip_Start_Date (void);
        ooInteger Mile_Get_Latest_Slip_End_Date (void);
        ooInteger Mile_Get_Cumulative_Slip_Start_Date (void);
        ooInteger Mile_Get_Cumulative_Slip_End_Date (void);
};      // ooMilestone
// The Breadth of Testing object, ooBreadthofTesting, is specific to the module object, ooModule, and contains attributes
// specific to the Breadth of Testing metric
class ooBreadthofTesting : public ooDBClass {
        ooHandle(ooDBClass) BreadthParent;
        ooReal BreadthCoverage;
        ooReal BreadthTestSuccess;
        ooReal BreadthOverallSuccess;
        friend class ooModule;
// ... and other definitions
};
// The Depth of Testing object, ooDepthofTesting, is specific to the module object, ooModule, and contains attributes
// attributes specific to the Depth of Testing metric
class ooDepthofTesting : public ooDBClass {
        ooHandle(ooDBClass) DepthParent;
        friend class ooModule;
// ... and other definitions
};
```

Figure 7.4: Partial Schema Definition Using Object-Oriented Data Model

```
// the Schedule object, ooSchedule, gives the schedule attributes relating to planned and actual start and end dates of the
// module or milestone specified in the database handle object.
class ooSchedule : public ooDBClass {
        ooHandle(ooDBClass) SchedParent;
        ooDate SchedPlanned_Start_Date;
        ooDate SchedPlanned_End_Date;
        ooDate SchedActual_Start_Date;
        ooDate SchedActual_End_Date;
        friend class ooMilestone;
};      // ooSchedule
//
// The following is a class specification for a module object. An ooModule object has one parent (except for the root
// module), zero or more children, a module name, a module identifier, and other attributes such as the Breadth of
// Testing, Depth of Testing, etc. It also has associated methods to return answers to queries relating to the module,
// constructors to create new modules or retrieve existing modules, etc.
//
class ooModule: public ooDBClass {
        ooHandle(ooModule) ooModParent;
        ooDBContainer(ooHandle(ooModule)) ooModChildren;
        ooString ModName (80);
        ooInteger ModId;
        ooBreadthofTesting ModBreadthofTesting;
        ooDepthofTesting ModDepthofTesting;
public:
        ooInteger Mod_Get_All_Test_Coverage (void);
        ooInteger Mod_Get_All_Test_Success (void);
        ooInteger Mod_Get_All_Overall_Success (void);
};      // ooModule
```

Figure 7.4: Partial Schema Definition Using Object-Oriented Data Model

Since the object model can capture behavior, it is possible to provide support for recursion by developing the appropriate methods. An object can comprise methods as well as

other objects which may either be of the same type or of different types. Take, for example, our recursive MPMS query:

How thoroughly has the software project XYZ been tested?

Given the OO schema definition as in Figure 6.2, the query would be posed in a manner similar to the following:

1. Look up the container class **MetModule** the global database **ooMetricsDB** using the handle given when the database was opened for the module matching name "XYZ". Call the pointer to this module **xyz**;
2. Obtain the values of the data members **Mod_Get_All_Test_Coverage**, **Mod_Get_All_Test_Success**, and **Mod_Get_All_Overall_Success** for from the ModBreadthofTesting object of module object **xyz**;
3. For each of the objects in the container class **ooModChildren**, obtain the value of the equivalent fields in the child modules.

7.3.2 Benefits of O-O Model for Metric Queries

The O-O model appears to meet the requirements of software metrics applications; supporting a wide range of queries from simple queries, temporal queries, and more complex recursive queries. It naturally represents complex and irregular relationships. Moreover, the tight integration of the OO model with OO programming language interfaces, as illustrated in C++ in our example, enables designers to smoothly integrate non-DBMS data sources and applications. This is especially important in MPMS applications where project management tools are not entirely centered around the DBMS.

7.3.3 Drawbacks of O-O Model for Metric Queries

A unique and powerful feature of the object-oriented model is a seamless integration of the data model and the meta model. Essentially, the meta model classes like 'object', 'class', etc. as well as application specific classes like 'requirements', 'code', 'tests', etc. can exist in the same hierarchy. This makes it very easy to adapt the object model to suit the needs of any

application. However, while the object model allows one to define application specific classes and integrate them with the generic ones, *it is still the responsibility of the application developer to design and develop the classes to be added to the class hierarchy.* Thus, we believe that the generic object model is not ideally suited for modeling a metrics database, since such a model would provide built in support for concepts in software metrics. Of course, a good implementation approach is to take a metrics data model and implement it as a set of classes using some object-oriented model and attach it to the basic object hierarchy.

Another severe hurdle in the use of object data models for database design stems from their relative novice status as compared to the relational model. While the latter has a well defined (and well accepted) standard, those for the former are still emerging. Consequently, the relational model enjoys the existence of (and experience with) several well-established database design, development, and maintenance methodologies and tools. These make the life of a relational database designer easier, eliminating housekeeping chores and allowing the designer to focus on the creative components. In this regards, there are several hurdles in the path leading to object orientation. The lack of a consensus on what OO methodologies should provide has made it difficult to come up with a standard in the industry. It has also hampered the development of commercial database design and development tools as vendors are hesitant to invest in a methodology which is not standard and thus may not get wide acceptance.

There are insufficient robust tools in the market that support object-oriented software development. Most vendors of existing OO software development tools have with insufficient market shares and questionable futures. Few MPMS applications in industry have used OODBMS despite its advantages.

7.4 Software Metrics Queries Using the Graph Data Model

Although the relational model has made prominent contributions in the research on database systems, applications have questions that cannot be answered within the relational model. Research on alternatives to the relational data model is motivated by the latter's disadvantage of forcing the stored data to have a flat structure that real data does not always have. This has motivated a considerable research in the past decade on structured data models, such as the semantic data models, graph data models, nested relations, and complex objects. Graves [GRA95] proposed a *graph-theoretic data model* for genome map databases which store the rapidly growing amount of mapping data contained in both published maps and laboratory notebooks, where graphs are defined as a collection of nodes and arcs and can represent genomic objects and relationships between them. Levene [LEV93] developed a graph-based data model called the *hypernode model* whose single data structure is the hypernode, a directed graph whose nodes may themselves reference further directed graphs. Aman [AMA92] presents *Gram*, a graph data model and query language where data is organized as graphs, and regular expressions over the types of node and edge labels are used to qualify connected subgraphs. Gram also includes an algebraic language based on these regular expressions and supports a restricted form of recursion.

The *Graph Data Model* (GDM) and its data language, the *Graph Data Language* (GDL) provides a solution to such applications by formulating schemas as directed graphs and data operations as algebraic operations on these graphs. A query is expressed as a path on a connected graph. Record types on a path may be recursively qualified by other paths. This approach gives the system the necessary guidance for determining access paths, thus reducing execution overhead of the GDM-based database management system. GDL also allows dynamic creation and deletion of link types, resulting in data independence since the user can redefine record types and links as a schema evolves.

7.4.1 Sample Software Metrics Queries in GDL

A partial schema that contains the necessary information to answer the metrics queries discussed for the relational data model is given in Figure 7.5:

```
define schema METRICS_DATABASE
        record MODULE
                attribute MODNAME                character      80
                attribute MODID                  integer
                search key MODNAME
                unique key MODID
        record SCHEDULE
                attribute MILESTONE              character      80
                attribute PLANNED_START_DATE     date
                attribute PLANNED_END_DATE       date
                attribute ACTUAL_START_DATE      date
                attribute ACTUAL_END_DATE        date
        record BREADTH_OF_TESTING
                attribute MODID                  integer
                attribute COVERAGE               real
                attribute TEST-SUCCESS           real
                attribute OVERALL-SUCCESS        real
        record DEPTH_OF_TESTING
                attribute MODID                  integer
                ... other attributes...
        ... other record definitions ...
        link childmodule (MODULE, MODULE)
        link has1 (PROJECT, MILESTONE)
        link has2 (MILESTONE, SCHEDULE)
        link include1 (MODULE, BREADTH_OF_TESTING)
                virtual MODID=MODID
        link include2 (MODULE, DEPTH_OF_TESTING)
                virtual MODID=MODID

        ... other link definitions ...
```

Figure 7.5: Partial Schema Definition Using Graph Data Model

Recursive queries such as *How thoroughly has module XYZ been tested?* can be performed using the transitive link operator \\ :

VARIABLE MODULE M1, M2;

path M1[NAME="XYZ"] → \childmodule\ → M2;

7.4.2 Benefits and Drawbacks of Graph Data Model for Metric Queries

The graph data model is excellent for handling recursive and complex queries, that are not handled by the relational model. The mathematical notation is, however, more elaborate and thus may have a steeper learning curve for the user, compared to the relational data model. Another important factor to compare is the implementation efficiency of a data model in commercial systems. GDL, like other graph-oriented data models, is still in a nascent stage as far as commercial acceptability is concerned. Thus, at this point its implementation is less than desirable from an efficiency viewpoint. This is mainly due to the expensive computation associated with traversals, especially recursive ones. However, providing an efficient implementation of GDL (or any other graph-oriented data model) in our opinion is a matter of engineering effort and financial investment, rather than a scientific hurdle. Extensive research has been done in the area of deductive databases [MIN88], especially on the evaluation of recursive predicates [BAN86], which can be used to implement graph traversals efficiently.

It is interesting to note that despite their obvious advantages over relational system for many applications, even commercial object-oriented databases are not up to speed with their relational counterparts, and are thus often not used. Evaluating traversal predicates in GDL is very similar to some navigational aspects of query processing in object data models. Hence, we believe that efficient techniques developed by the object database community can be adapted for efficient implementations of GDL.

7.5 A Formal Approach for Semantic Modeling of Metrics Data for Quality and Risk Management

In order to support high level notions of quality and risk and to address complex queries which not only span temporal dimension but also relate data across metrics, we need to develop a formal data modeling framework. The objective is to provide an environment for the user to express complex semantics without putting any constraints which may be exhibited by the underlying data model [PAU97]. For this purpose, we propose a two-pronged approach illustrated in Figure 7.6(a). First, we introduce a formalism for temporal modeling of data using a set of generalized *n*-ary temporal relations. Second, we introduce an object-oriented model to store and retrieve semantics associated with software quality and risk. The top flow corresponds to temporal modeling and characterization of scenarios present in the metrics data. The bottom flow corresponds to the users' view of the data where grouping/linking of information across metrics and projects is supported using an object-oriented paradigm. The integration of information within a metric and across metrics leads to an efficient mechanism for on-line query processing for software management. For most of the queries, the proposed framework can avoid performing computation on raw data during query processing since such computations can be quite extensive and can be carried out off-line. Practically this framework allows the conceptualization of metrics data using both bottom-up as well as top-down object-oriented approaches for data abstraction. In the bottom-up approach, a user can build complex scenarios using simple events while in the top-down approach, a user can integrate/group scenarios/projects having identical behavior.

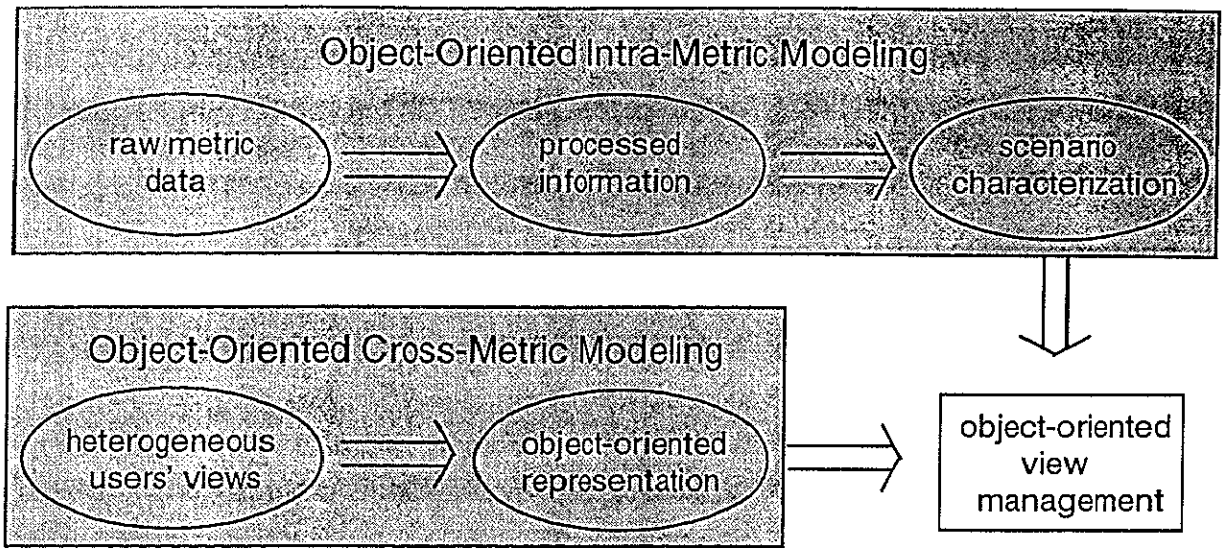

Figure 7.6 (a): Object-oriented Abstraction for Management of Heterogeneous Views

7.5.1 A Two-Level Spatio-Temporal Modeling Approach

We can view semantics of metrics data in a two domain; space and time, as depicted in Figure 7.6(b). Spatial data modeling and querying deal with those semantics which do not involve any time interval or duration. Such semantics deal with conventional types of retrieval (such as a SQL in a relational mode). Queries such as finding the number of CSCI's in a system, the start date of a project, expected resource utilization by the software etc., fall into this category. Most of the discussion in the previous sections deals with this type of domain. In temporal domain we concentrate on semantics dealing with time and history of data, and have non-zero duration associated with it. Example queries related to temporal domain are given in the Section 7.5.1.2.

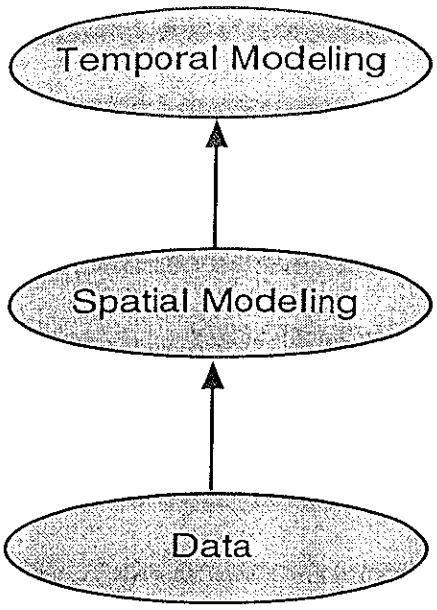

Figure 7.6(b): Two-Level Modeling of Metrics Data

7.5.1.1 Modeling in Spatial Domain - Low Level

For representing semantics associated with spatial (non-temporal) data we can use an object-oriented design discussed in the previous sections. One of our objectives is to develop visual query representation so that more complex queries can be easily formatted. For this purpose we plan to use the approach proposed in [MOH93]. In order to facilitate the querying process, in the next section we present a framework for visual query functions using the temporal operations discussed in earlier section. The overall query formulation consists of a non-temporal part and a temporal part. In this approach, three basic graphical primitives are used which are shown in Figure 7.7. A *class icon*, used for referencing any particular class in the schema, is shown in Figure 7.7 (a). A database class name is specified in the top portion of the *class icon*, while the bottom portion is used to reference any particular instance of the class. Figure 7.7 (b) shows an *attribute icon*. The top portion (*label portion*) is used for specifying the attribute name while the bottom portion is reserved for specifying conditions on the attribute.

Typically a condition can be either (i) the value of the attribute, (ii) the range of values of the attribute, or (iii) a constraint on the value of the attribute. Such conditions are identical to the ones used in SQL or GDL based queries. The *link icon*, shown in Figure 7.7 (c), refers to the relationship between classes. The top, middle, and bottom portion are used to specify type of the link, link class name, and instances of a particular link, respectively. Such a visual interface can specify a rich set of queries, including selection, restriction, join, queries involving reflexive relationships, universal quantification, negations, intensional queries (querying of schema), and transitive closure, among others.

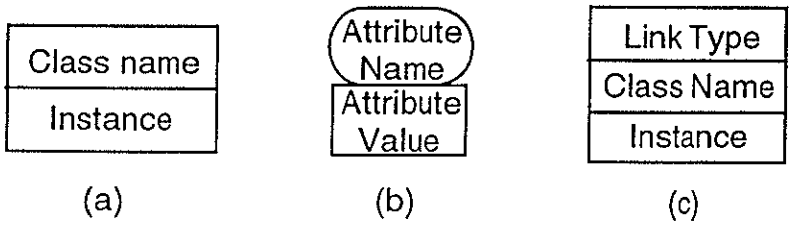

Figure 7.7: Visual query primitives

7.5.1.2 Modeling in Temporal Domain - High Level

In order to model temporal semantics of software data, and to formally express complex queries involving low level information discussed above and temporal semantics, we first need to introduce the notion of a scenario [PAU97]. A scenario represents a state of some behavior of the development process that persists for a period of time and is generally captured through the software metrics data. Examples may include prolonged durations of opening of priority 1 faults, the continuous slippage of schedules, manpower requirements continuously exceeding the expected requirements, requirements remain undefined or untraceable for a long duration, etc. These simple scenarios can be used to build more complex scenarios that can ultimately be

abstracted into the notion of quality/risk, and can be represented in the form of a multi-level hierarchy.

In order to express the scenarios in a formal manner, we use the concept of binary temporal relations. The basic component of a temporal relation is a temporal interval, which is a non-zero duration of time in any set of units. It is characterized by its two ends, or instants and is formally defined as follows:

Interval: *Let [S_T, ≤] be a partially ordered set, and let a, b be any two elements of S_T such that* $a \leq b$. *The set* $\{x | a \leq x \leq b\}$ *is called an interval.*

Given any two intervals, there are thirteen distinct ways in which they can be related. These relations show how two intervals relate in time; whether they overlap, abut, precede, etc. The thirteen relations can be represented by seven cases, since six of them are inverses. The seven relations are *before*, *meets*, *overlaps*, *during*$^{-1}$, *starts*, *finishes*$^{-1}$, and *equals* which are shown in Figure 7.8. For example, *after* is the inverse relation of *before*, i.e., *a after b = b before a*. Therefore, given any two intervals, it is possible to represent their relations by using only one relation with a possible change of the interval labels. Note that α *during*$^{-1}$ $\beta = \beta$ *during* α, α *finishes*$^{-1}$ $\beta = \beta$ *finishes* α. The constraints associated with these temporal relations are listed in Table 7.1.

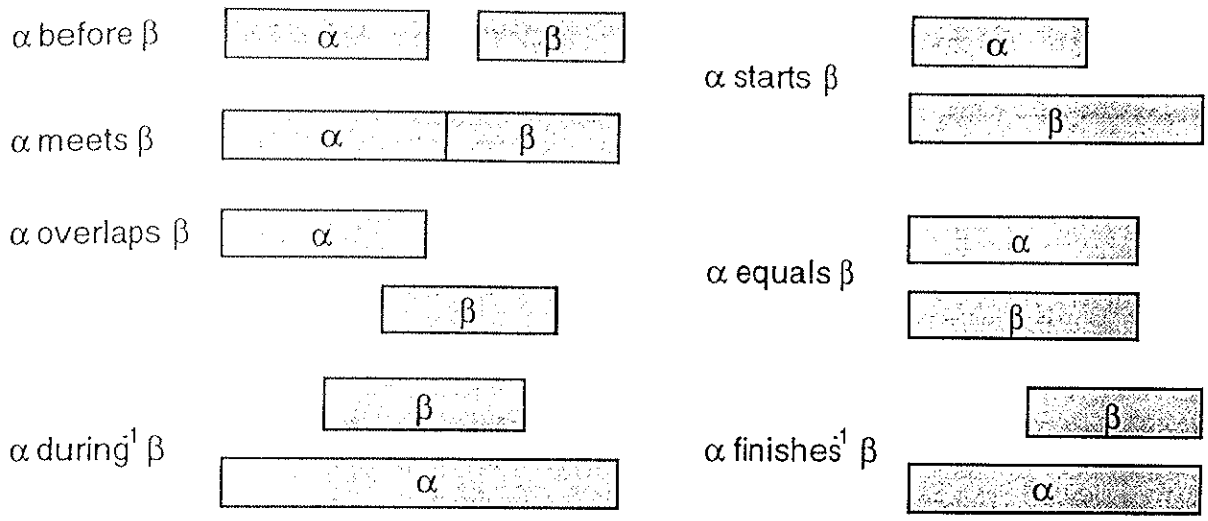

Figure 7.8: Binary Temporal Relations

Relation name	Symbol	Constraints
before	B	$\tau_i^e < \tau_{i+1}^s$
meets	M	$\tau_i^e = \tau_{i+1}^s$
overlaps	O	$\tau_i^s < \tau_{i+1}^s < \tau_i^e < \tau_{i+1}^e$
contains	C	$\tau_i^s < \tau_{i+1}^s < \tau_{i+1}^e < \tau_i^e$
starts	S	$\tau_i^s = \tau_{i+1}^s \wedge \tau_i^e < \tau_{i+1}^e$
completes	CO	$\tau_i^s < \tau_{i+1}^s \wedge \tau_i^e = \tau_{i+1}^e$
equals	E	$\tau_i^s = \tau_{i+1}^s \wedge \tau_i^e = \tau_{i+1}^e$

τ_i^s = starting coordinate of object τ_i, τ_i^e = ending coordinate of object τ_i

Table 7.1 Constraints with Binary Temporal Operators.

7.5.1.3 Identification of Scenarios from Metric Data

Temporal operators can be used to express temporal semantics among various metric data and the relative occurrences of scenarios. A scenario in the metrics can be identified and represented by a tuple containing an id and a set of temporal descriptions, (ρ= (oid,d_d,{s_t}), where * oid : object identifier for this scenario, * d_d : descriptive data, e.g., object type, name of metrics, etc. * {s_t}: an ordered set of temporal description, s_t = (π, τ, m_v); * π: the starting timing (program month) of the appearance of scenario; * τ: the duration of the scenario; $\tau = n\gamma + 1$, $n \geq 0$ and is an integer.

A property of scenarios is *concatenability*, which is the foundation for constructing a simple scenario. Before presenting the property itself, we introduce the semantics of predicate P. A predicate (event) $P(a_1,\ldots,a_m)$ (e) which is true during an interval i can be represented as P $(a_1,\ldots,a_m, i)$ $e(i)$). Concatenability means that if a condition on a scenario is true in intervals I and J, and I *meets* J, then the same condition is true in an interval $K = Meets(I, J)$. Formally, $\forall i, j\ e(i) \wedge e(j) \wedge M(i,j) \rightarrow e(M(i,j))$. An example is that if fault F specified by Fault Profile metric is open during intervals I and J, and I *meets* J, then F remains open during an interval $K=Meets(I,J)$.

When the scanning of metrics is over we need to perform concatenation of scenarios and include that in a set of object collections, OC. The procedure is as follows:

If an oid is unique, then put the corresponding tuple ρ in the object collection **OC**;

otherwise, perform concatenation as follows.

If $\rho_i.oid = \rho_j.oid$, then create a new tuple ρk,

where $\rho_k.oid = \rho_j.oid$, $\rho_k.d\_d = \rho_i.d\_d \cup \rho_j.d\_d$, $\rho_k.\{s\_t\} = \rho_i.\{s\_t\} \cup \rho_j.\{s\_t\}$.

Put ρ_k in **OC**. Note that within $\rho_k.\{s\_t\}$, if $s\_t_r = (\pi_r, \tau_r, m\_v_r)$, $s\_t_w = (\pi_w, \tau_w, m\_v_w)$, and

$\pi_r+\tau_r = \pi_w$, then create $s\_t_u = (\pi_u, \tau_u, m\_v_u)$, where $\pi_u = \pi_r$,

$\tau_u = \tau_r + \tau_w$, $m\_v_u = m\_v_r \cup m\_v_w$, put s_t_w in $\rho_j.\{s\_t\}$, remove s_t_r and s_t_w from

$\rho_j.\{s\_t\}$.

A concept can be illustrated using Figure 7.9, where x-axis represents time (program month) and an identified scenario is represented by a set of intervals in which it appears. From now on we will represent scenarios by O_1, O_2, O_3, and O_4 as identified in Figure 7.9. The two intervals where O_1 appears are related by *before* relation. Suppose five domain objects are

identified from later part of a metric, where two objects $O1$ and $O4$ have appeared in some earlier part of that metric.

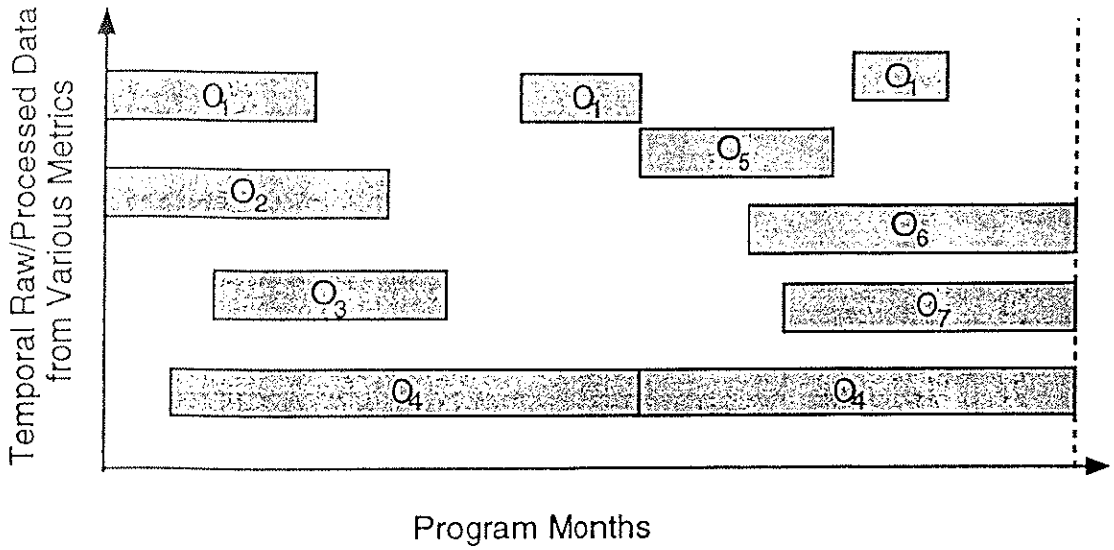

Figure 7.9: Example of Scenario Identification

From analysis vectors (*m_v's*) we can identify relative temporal relations among various scenarios (o_i's). In order to identify the duration of a persisting scenario, we introduce an operator (SI) which returns a set of intervals when it is applied to a software metrics. The proposed syntax is: *SI(x); x/table_name; qualification clause* Assume that there is a *time stamp* (program month) field associated with each tuple and each time stamp has the same granularity and is treated as one *interval.*

The processing steps associated with *SI* operator are described below.

- Evaluate the query *x; x/table_name; qualification clause* and a number of tuples are returned.
- Project the time stamp filed of the returned tuples.
- Sort tuples in an ascending order, using time satmo as the sort key.
- Concatenate intervals *meet* together.
- Record an ordered set of intervals related by *before* relation.

As an example, consider the following query to the cost metric: *SI(x); x*/COST; *x.vlaue* < 1000 ∧ *x.value* > 800 The query is processed as follows. Evaluate the query *x; x*/COST; *x.vlaue* < 1000 ∧ *x.value* > 800 and assume the tuples in Table 7.2 are returned. Project the field *Timestamp*. Sort of the timestamps in a buffer. The buffer after sorting is shown in Table 7.3. Concatenate intervals so we have three intervals (4/5/95,4/7/95), and (11/27/95,11/27/95). The interval information can be stored in a relation as shown in Table 7.4.

	Timestamp
...	4/6/95
...	4/5/95
...	4/7/95
...	10/11/95
...	11/27/95
...	10/12/95

Table 7.2: Query result

4/5/95
4/6/95
4/7/95
10/11/95
10/12/95
11/27/95

Table 7.3: Buffer

start	end
4/5/95	4/7/95
10/11/95	10/12/95
11/27/95	11/27/95

Table 7.4: Interval relation

7.5.2 A Petri-Net Based Spatio-Temporal Modeling of Software Metrics

Spatio-temporal modeling of software metrics is important for users to ultimately construct complex views or to describe scenario or notion of quality/risk, etc. These notions can be expressed by interpreting collectively the various scenarios present in the metrics data. In a simplistic manner, the quality/risk can be described by observing the total (or partial) duration during which a scenario appears in the data and its relative occurrence with respect to other scenarios over a given span of time. For example, viewing that a project is of high quality can be a high level scenario in a user's specified query. Modeling of this scenario (object) requires occurrence of multiple temporally related *sub-events* (scenarios). The overall process of expressing such a scenario requires a priori specification of multiple temporal sub-events. It can be noticed that a simple temporal event can be expressed formally as a logical compostion of various low level scenarios that span over part of the life cycle of software development process. Subsequently, more complex abstractions (scenarios) can be defined recursively in terms of other scenarios related by complex spaghetti of temporal relations.

In this section, we present a formal model for spatio-temporal semantic modeling of software metrics which provides a comprehensive and easy to understand representation of complex spatio-temporal semantics The model is based on the Petri net [PET77, PET81,

MUR89, AGE79] and is suitable for describing any temporal semantic associated with software metrics. Through out the remaining thesis, this model is termed as Software Metrics Petri Net (SMPN). We introduce and prove theorems about various properties of SMPN.

7.5.2.1 Temporal Semantic Modeling

An abstract model is sought which characterizes the scenarios/events presented in software metrics. Several issues arise. The most important one is to devise a mechanism which allows the specification of temporal relationships among events and to model them for subsequent query processing and retrieval. In the general case, a technique to model the real-world in a succinct way is desired. Several specification techniques are can be used including programming languages, real-time specification languages, and Petri nets. The graphical modeling techniques for time-dependent systems include including finite state machines (FSM), and Petri nets (PN). Finite state machines are a specialization of Petri nets and therefore are a subset of the more general Petri nets, and are not considered here. Other aspects of timing have also been represented with the Petri net, but not for temporal semantics modeling of data. These include EMPN, a technique for specifying real-time distributed systems [CHA88], PROT nets, for modeling process control systems [BRU86] Colored Petri nets for real-time control of manufacturing systems [JEN81, KAS88], Timed Petri nets for real-time system timing requirements [COO85].

7.5.2.2 Petri Nets

The primary features of a Petri net are summarized as follows [CHA88]:

(1) It allows representation of concurrent, asynchronous, and non-deterministic activities,

(2) it provides capability for decomposition into multiple or hierarchical graphs, and

(3) it is supported by many types of analysis on both the graph structure and dynamic characteristics [MUR89].

Petri nets can be used in a specification and modeling tool for temporal specification of metrics data. Also, Petri nets provide a convenient graphical representation and are amenable to analyses including Markov process modeling [MUR89, HOL87, ZUB85].

Temporal modeling of software metrics data is inherently a task which assumes parallel and sequential activities. For software metrics data, the parallelism is in various types of metrics collected concurrently. In this section, we propose to use Petri nets for semantic modeling of metrics data. More specifically, we consider the timed [RAZ84, HOL85, ZUB85], and augmented [COO83] Petri net models. These models are chosen for their desirable attributes of representation of concurrent and asynchronous events.

In general, a Petri net is defined as a bipartite, directed graph N=(T, P, A) where [HOL85],

T = {t 1 ,t 2 ,...t n } is a set of transitions (bars)

P = {p 1 ,p 2 ,...p m } is a set of places (circles)

A: {T x P} U {P x T} -> I, I = {1,2,...} is a set of directed arcs (arcs).

A marked Petri net N' = {T,P,A,M} includes a marking M which assigns tokens (dots) to each place in the net:

M: P -> I, I = {0,1,2,... } is a mapping from the set of places to the integers.

For simple Petri nets, the time from enabling a transition to firing is unspecified and indeterminate. Firing of a transition is assumed to be an instantaneous event. To represent the concept of nonzero time expenditure in a Petri net, extensions of the original model are required. A class of enhanced Petri net models have been developed which assign a firing duration to each transition [RAZ84, HOL85, ZUB85]. These models are generally called timed Petri net (TPN) models, and map well to Markov performance analysis.

Another TPN model [COO83] represents processes by places instead of transitions. Nonnegative execution times are assigned to each place in the net. With this scheme the notion of instantaneous firing of transitions is preserved, and the state of the system is always clearly represented during process execution, i.e., tokens are at all times in places, not transitions. This "augmented" model has the advantage of compactness of representation. Either process timing scheme can be used for our purposes, however, we choose the more compact of the representations.

7.5.2.3 The Proposed Petri Net Model for Semantic Modeling

We supplement the augmented model with resource information associated with TPN models [RAZ84], for the purpose of illustrating the use of the metrics events. This modification of earlier Petri net models result in a new model called Software Metrics Petri Net (SMPN) for distinction from other PN models [PAU97]. The SMPN augments the conventional Petri net model with the event duration conditions attached to the places. These events are spatial semantics represented as visual icons. Therefore, an SMPN is described as,

$N_{SMPN} = \{T, P, A, D, M\}$

This specification consists of a set of places (P), a set of arcs (A), and a set of transitions (T). M represents marking of tokens. An arc connects a transition to a place or vice versa. Each place has exactly one incoming arc and one outgoing arc. Additionally, each place has the following attributes:

- $S : P \rightarrow$ *String, is a mapping from a place to a symbolic representation corresponding to the source of the interval. This source is the embedded non-temporal semantics icon defining the scenario associated with the interval.*
- $\psi(d) : P \rightarrow \psi$, *is a mapping from a place to the user-specified*

duration expression. ψ is either (i) d, (ii) d = number, (iii) d = d_v,

or (iv) a range expression specifying the allowable duration associated with d,

where d and d_v are duration variables representing durations of intervals,

arithmetic operators <, >, <=, >= can be used.

- *A delay place has no symbolic representation.*

7.5.2.4 Query Evaluation and Execution of SMPN

Associated with the definition of the Petri net is a set of firing rules governing the semantics of the model. Since we define a transition to occur instantaneously, places rather than transitions have states. The firing rules for the SMPN govern its execution and results in the processing of queries modeled by the SMPN. These rules are as follows:

(1) A transition t_i fires immediately when each of its input places contain an *unlocked* token.

(2) Upon firing, the transition t_i removes a token from each of its input places and adds a token to each of its output places.

(3) After receiving a token, a place p_j executes the low level spatial query check for the duration condition associated with the scenarios/events represented by the low level spatial sub-query. In case the condition is true, the token is *unlocked.* Otherwise, it remains *locked* which causes the failure of the execution of query. In other words, the query is not successful.

7.5.2.5 Examples of SMPN

An example elaborating the SMPN model is given in Figure , where five scenarios, S1 to S5, and two delay places, are represented. In this case, S1's duration is less than 10 units of time (e.g., program month), followed by scenario S2 with duration greater than or equal to 5, less than or equal to 7 units of time, followed by S3 with duration 6 units of time. S4 starts two to five units later than S1 and has a duration less than or equal to 3 units of time. Similarly, Scenario S5

starts four or more units of time later than S1 and has a duration of five units of time. As can be seen, the proposed methodology is straightforward yet powerful.

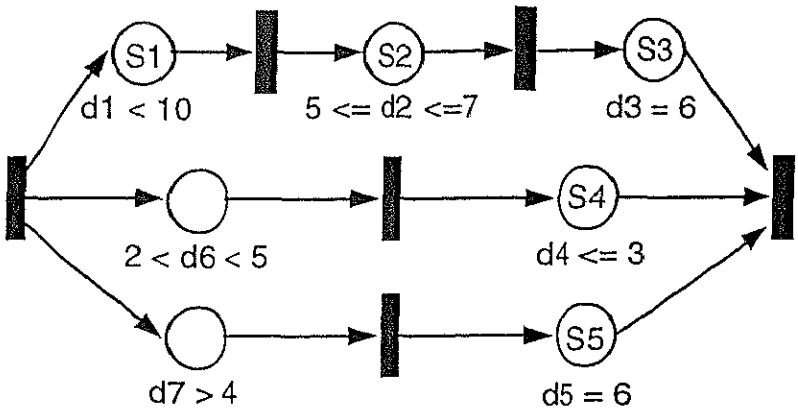

Figure 7.10: Example of a SMPN

An example involving both temporal and non-temporal part of the querying aspect is shown in Figure 7.11. The query is: ``Does the system integration of system with name 'MDBMS' consist of two scenarios `completion of requirement traceability' followed by 'completion of testing phase' with a delay between one to two months?" The Figure 7.11(a) specifies for each class (requirement traceability or testing) the intervals during which the system named `MDBMS' was being developed, and all the intervals generated through SI operator are represented as an aggregate interval (by specifying ∀). The Figure 7.11(b) denotes that the two aggregate intervals are one after the other with a delay of one to two units of time (month in this case).

Figure 7.12 describes an example of scenario describing various phases in software development life-cycle. The associated SMPN in fact can be used to describe the dynamics of the development process.

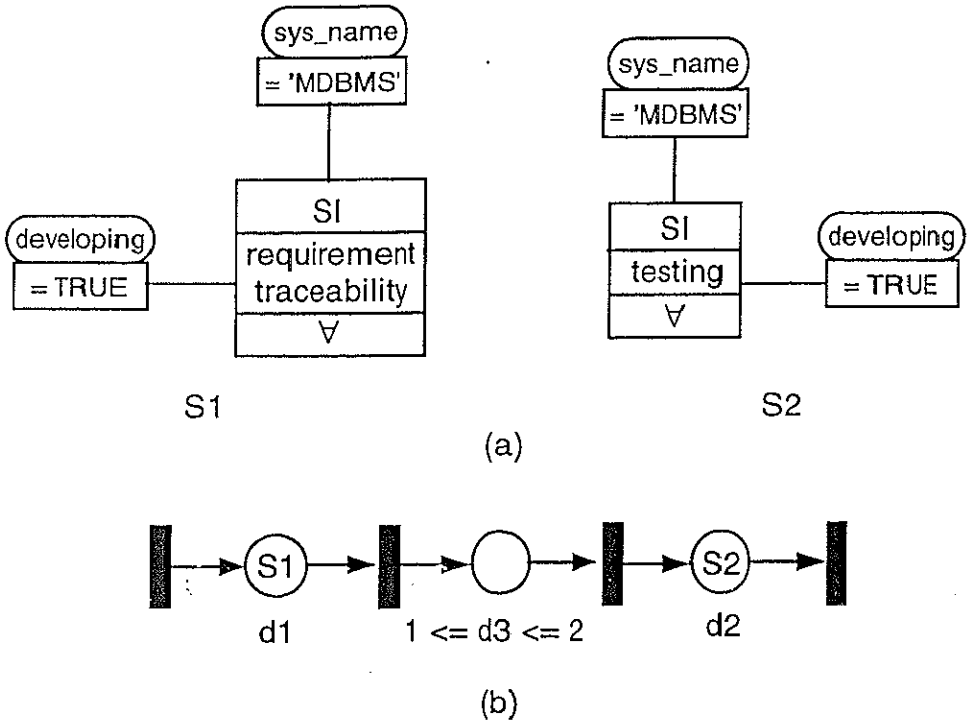

Figure 7.11: Example of a SMPN involving embedded non-temporal semantics

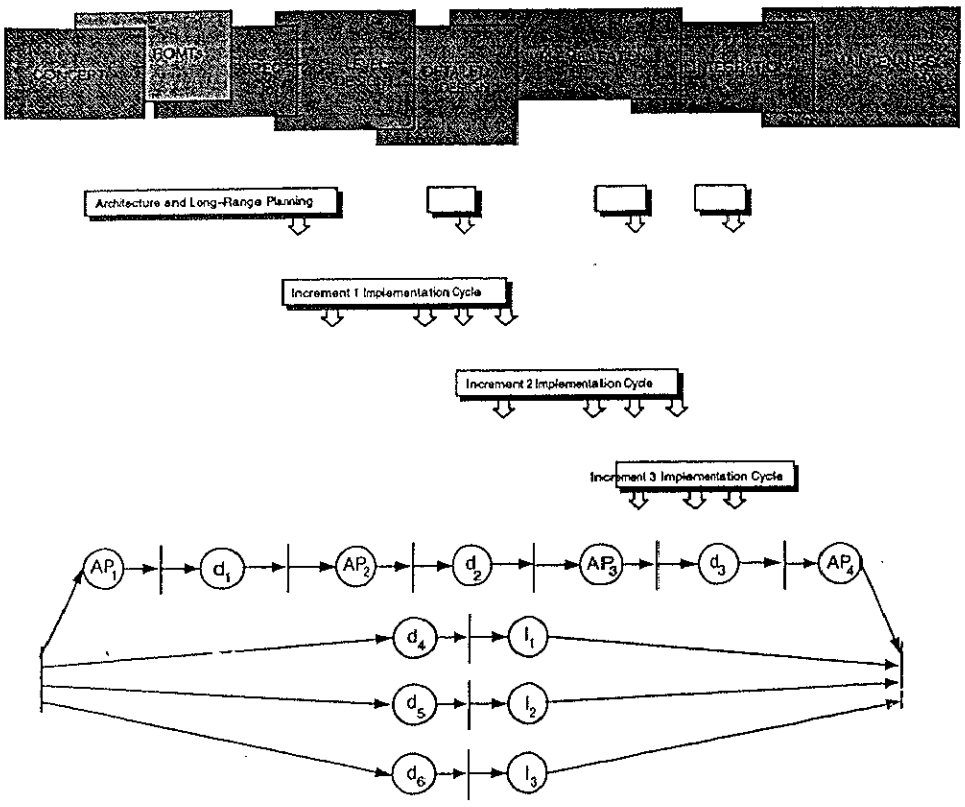

Figure 7.12: A Scenario Describing Various Phases in Software Development Life-Cycle

7.5.2.6 Representing Standard Database Query Operations using SMPN

We now briefly elaborate how the following database operations in temporal dimension [ISH95] can be expressed using the proposed SMPN.

(1) Tintersection(I1,I2) returns parts of I1 and I2 which temporally intersect.

(2) Tdifference(I1,I2) returns a part of I1 which do not temporally intersect with I2.

(3) Tunion(I1,I2) returns I1 and i2 ordered in time with possible overlaps.

(4) Tselect(I1,P) returns a part of I1 which satisfies P.

(5) Tjoin(I1,I2,P) = Tselect(Tjoin(I1,I2),P).

(6) Tproject(I1,Func) returns the result of applying function Func on I1.

Operations (1) - (3) and (5) are represented by Figure 7.13(a) and operations (4) and (6) are represented by Figure 7.13(b).

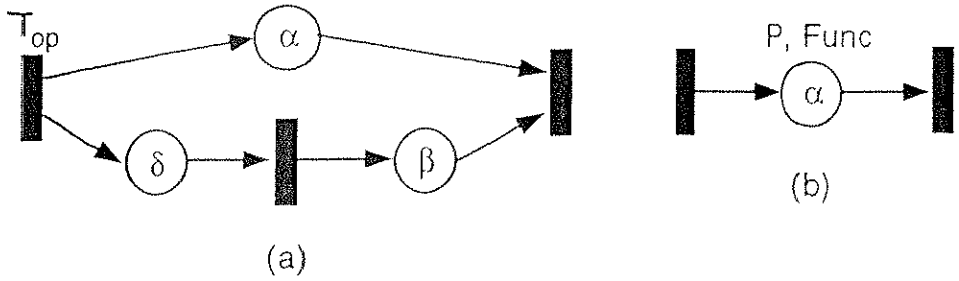

Figure 7.13: SMPN based representation of standard database operations.

In summary, the proposed SMPN model and the algorithms for identification of scenarios can provide a powerful environment for constructing and characterizing arbitrarily complex scenarios. The model allows development of hierarchical relationships among different types of complex scenarios that inherit properties of simple events. Another important feature of SMPN is that it can allow visual querying of scenarios and semantics. Its graphical characteristics is intuitive and natural for visual representation and formulation of spatio-temporal semantics [PAU96a, PAU96c].

7.5.2.7 Basic Temporal Relations and the SMPN

Earlier, in Figure 7.8 we have shown thirteen temporal relations. These intervals can be described as processes $P\alpha$ and $P\beta$ with durations $\tau\alpha$ and $\tau\beta$, respectively. We say a process is atomic if it cannot be decomposed into subprocesses. The following theorem relates temporal intervals to the SMPN.

Theorem 7.1: Given any two atomic processes specified by temporal intervals, there exists an SMPN representation for their relationship in time.

Proof: We show perfect induction by constructing a Petri net representation for each temporal interval by assigning a place for each process with associated resource and duration. A delay process and place is introduced depending on the type of tempo ral relationship. Let $P\alpha$, and $P\beta$ be atomic processes with finite, nontrivial (nonzero) temporal intervals, and $\tau\alpha$ and $\tau\beta$, respectively. Let γ be the finite delay duration specific to any temporal relation T R . We can easily construct a Petri net for each relation.

For the *meets, starts*, and *equals* relation, no delay place is required, i.e., $\gamma = 0$. For the *overlaps, during*, and *starts* relations, these model do not explicitly indicate the delay of the process which completes first. Since both places (processes) lead to the same transition, the Petri net requires that tokens from each be unlocked prior to the execution of the succeeding process. Thus the Petri net ensures synchronization for these cases.

According to Theorem 7.1, for any two atomic processes and their temporal relation there is a corresponding Petri net (SMPN) model. The converse is true as well, that is, for any SMPN model a corresponding temporal relation can be uniquely identified.

Theorem 7.2: An arbitrarily complex process model composed of temporal relations can be constructed with Petri nets by choosing pairwise, temporal relationships between process entities.

Proof: Follows from the proof of Theorem 7.1.

7.5.2.8 Properties of the SMPN

Using the SMPN models of Theorem 7.1 and the construction process of Theorem 7.2, it is clear that no modeling of Petri net *conflicts*. Stated differently, places of SMPN models have exactly one incoming and one outgoing arc. Therefore, SMPNs are a *marked graph* [COM71],

and possess their characteristics. The following is a discussion of some important properties of SMPNs.

7.5.2.8.1 Precedence Relationships and Partial Ordering

An SMPN indicates precedence relationships among scenarios. These relationships can be a partial or strict order, depending on the type of temporal relationship modeled. For sequential relations, the precedence relation is strict, since no reordering is possible. For concurrent relations, interval durations can be identical for a set of scenarios, and therefore a partial order exists.

7.5.2.8.2 Reachability

SMPNs are deterministic because no conflicts are modeled, transitions are instantaneous, and tokens are remain at places for known, finite durations. The result is a linear sequence of states indicated by the reachability tree for an SMPN.

7.5.2.8.3 Liveness

Liveness implies the absence of deadlocks. A Petri net is live if, for all markings M in the reachability set of the initial marking, it is possible to fire any transition through some progressing firing sequence [PET77]. The SMPN models presented do not indicate any tokens and are therefore not live. To initiate an active SMPN process we can include an initial place with a single token in the SMPN model. By terminating the SMPN at the initial place we create a cyclic presentation rather than a strictly linear one. The result is a strongly connected graph, that is, one where there exists a directed path from every node to every other node. Since the reachability tree is linear and the SMPN construction process does not introduce any cycles, by introducing a single feedback loop (cycle) an SMPN is decidedly live. Additionally, liveness implies complete *coverability* for all M in the initial marking.

7.5.2.8.4 Boundedness

A Petri net is *k-bounded* if the number of tokens at each place in a net does not exceed some k for any marking reachable from the initial marking [PET77]. If the net is 1-bounded, it is called *safe*. Since SMPNs are built with hierarchical composition process without cycles, for every transition with q outgoing arcs there exists a corresponding transition with q incoming arcs, therefore, k-boundedness is assured. For every token created at a forking arc, one is absorbed at the joining arc, and the k tokens is preserved. Since $k = 1$ for SMPNs, they are safe.

7.5.2.8.5 Conservation

A Petri net is conservative if the number of tokens in the net does not decrease. For the SMPN, the number of tokens varies with the number of concurrent scenarios, and therefore is not conservative. However, the initial and final number of tokens always remains equal to one, as discussed.

7.5.3 Object-Oriented Modeling of Software Quality/Risk

Considerable semantic heterogeneity may exist among users of software metrics data due to the differences in their pre-conceived interpretation or intended use of the information. Semantic-based integration of different views may be required for a large number of users. Data management and efficient query processing, in the process of such integrations, is a complex and challenging problem.

Conventional data modeling techniques (such as relational model) lack the ability of managing complex scenarios of software metrics and supporting heterogeneous views of the data. For example, as mentioned earlier, the relational model has a drawback of losing semantics which can cause erroneous interpretation of views and events. The object-oriented technology, on the other hand, can provide a powerful paradigm to meet the requirements of our semantic modeling and management of complex metrics data. Its data and computational encapsulation features

offer elegant data modeling capabilities at various levels of information granularity. The paradigm can allow users to combine multiple views of the data into a single comprehensive view. The basic concept that data is associated with procedures manipulating it, is especially appealing in modeling metrics data, where the raw data needs to be analyzed and temporal contents of the information needs to be combined using rather complex logic. Object-oriented modeling allows such complexity to be independently managed and linked together via communication among objects using messages. The *class* concept in object-oriented paradigm is especially suitable for semantic-based grouping of events. In this section, we propose a modeling process of multiple and heterogeneous views of users in an object-oriented environment and describe how multiple scenarios can be integrated into a single framework. The fundamental premise here is that we can establish a correspondence between hierarchical relationship of scenarios discussed in the previous section and different classes of objects using various object-oriented abstractions. In object-oriented environment, objects, classes, and meta classes can be defined recursively at arbitrary levels. By embedding a scenario in a class, not only necessary information of a scenario can be recorded as attributes, but most importantly, the identification of a scenario can be treated as method(s) of a class. Also, if an attribute is another scenario (class), the current class can invoke the methods of that class by sending message(s) to it.

7.5.3.1 From Scenarios to Objects

For mapping of scenarios to classes, scenarios can be categorized into two generic classes. A *generic persistent scenario* and *a generic temporal scenario.* In Table 7.5, we provide a generic template for declaring a *persistent scenario* class. Along with the attributes, such as object id, pointer to object definition, object id list of participating metrics, etc., the main component of the class is the *class method* to identify the given scenario. The actual scenarios identified are the instances of a persistent scenario class. Since a scenario definition has

parameters, the identification procedure is performed for each combination of parameters. The system is updated with new instances and scenarios types as they are identified, during the database loading process (or while mining an archival database). Note that the identification procedure is only associated with the class definition, i.e., it is a class method, not a method of an instance of a class. Additionally, the *duration attribute* is used to record the *persistence* of a scenario. The method *return_single_value_attribute* is used to return any attribute with a single value.

CLASS *Generic (Persistent) Scenario*
ATTRIBUTE
object_identifier (oid)
scenario_definition_expression* (class method)
metric, starting_program_month
duration, *TAG*
oid_list_of_participating_object*
oid_list_of_component_scenario*
METHOD
identification_procedure() /*class method*/ (For each tuple-collection representation of a metric)
Perform Algorithtm to identify the (persistent) scenarios
return_single_value_attribute(attribute_name)
return_participation_object_id()
return_component_event_id()

Table 7.5: Generic Scenario

Table 7.6 provides a template for declaring a *temporal scenario* class. For an instance of this class a component can be an instance of another temporal scenario class or may be an instance of a persistent scenario class with the property that instances of some low level classes are related with the temporal relation *meets*. The structure of the temporal scenario class is similar to that of *persistent scenario* class, except that the identification procudure is different.

CLASS *Generic (Persistent) Temporal Event*
ATTRIBUTE
object_identifier (oid)
event_definition_expression* (class method) metric ID, starting program months;
duration,
TAG oid_list_of_participating_object*
oid_list_of_component_event*
METHOD
identification_procedure() /*class method*/ (For each metric/
Perform Algorithm 2 to identify the temporal events
return_single_value_attribute(attribute_name)
return_component_event_id()
return_participation_object_id()

Table 7.6: Generic temporal event Abstraction of Software Metrics Data in Object-Oriented Paradigm

Considering scenarios as objects classes, in this section we show how existing object-oriented abstractions can be used to define new classes and how inheritance can be used to construct complex views. An important aspect of these abstractions is that they allow grouping

and merging of information entities which may not have any temporal relationship among them but some general semantics.

For developing an abstraction of software metrics data, we can use either a top-down (specialization) or bottom-up (generalization) approach. Irrespective of the abstraction used, we can utilize three types of semantic relationships between classes. These include, generalization (IS-A), aggregation (IS-PART-OF), and temporal aggregation. They are used together depending on the grouping requirements of users. However, there are some rules that must be observed while using these abstractions for modeling of metrics. The view hierarchy consists of three types of abstractions, namely, generalization, aggregation (IS-PART-OF), and temporal aggregation (n-ary). The connection of nodes should be meaningful. Figure 7.14 depicts the overall process of abstraction via aggregation. Each place represents an SMPN except the spatial objects which are embedded visual icon based non-temporal semantics.

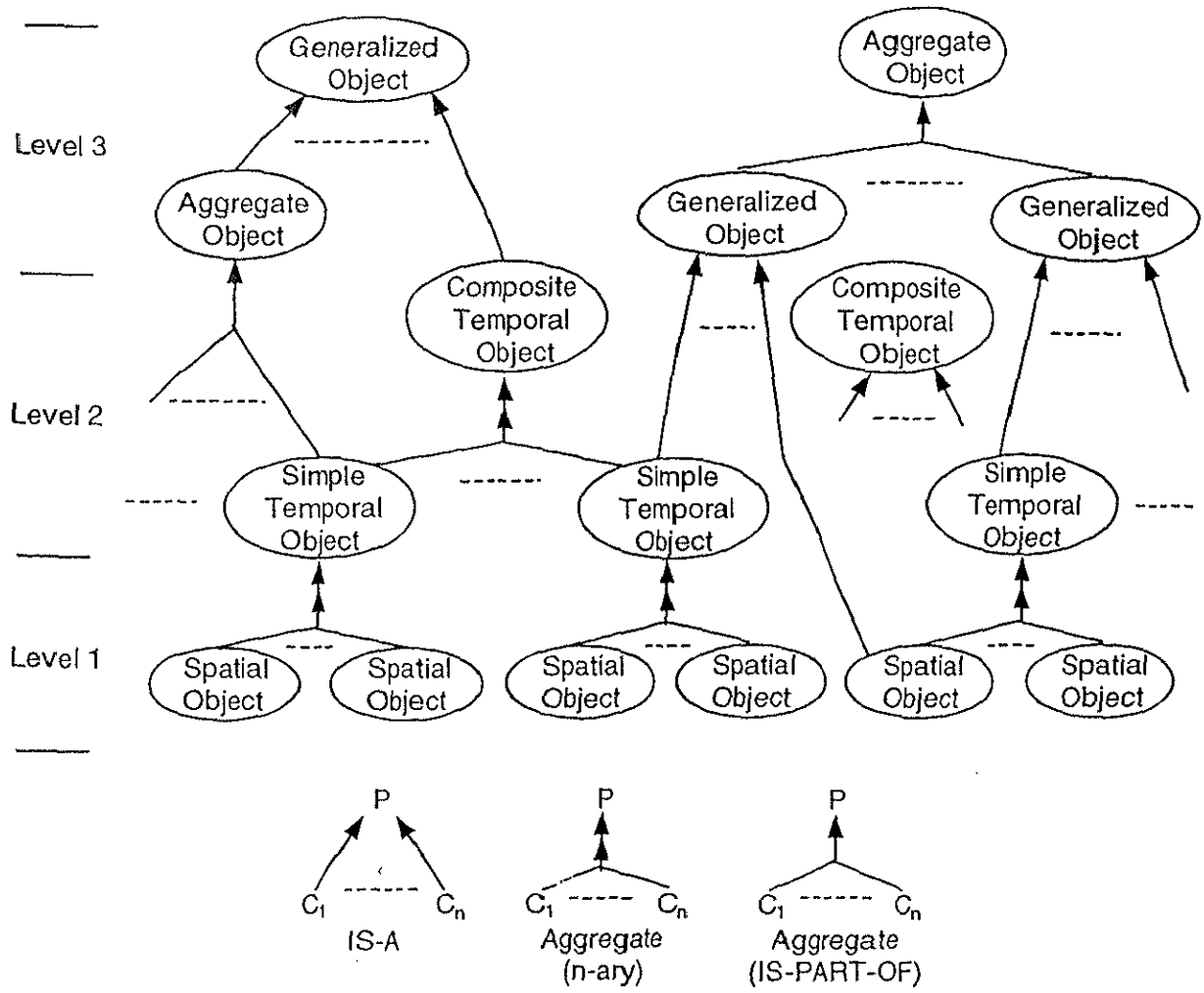

Figure 7.14: Object-oriented Abstraction using SMPN

7.6 Performance Considerations for Implementation of Software Database Management System

In this section we evaluate the performance of the relational and graph based data management system in terms of a possible implementation of the type of object-oriented framework discussed in the previous section. Benchmarks are used to quantify the relative performance of different systems to evaluate their effectiveness/worth for a certain type and amount of workload. Since the cost of implementing and measuring a specific application on many (if not all available) systems may usually be much larger than the price of the systems,

generic benchmarks are used to as a rough estimate of one system performance over another. However, no single metric can measure the performance of all computer systems for all the various applications and the permutations of all possible tasks within them. System performance may vary widely from one application to another since each system may be optimized for certain application. Therefore we need domain specific benchmarks, characterizing typical applications in a specific problem area. For a domain specific metric to be useful it must be relevant to that problem domain. It addition it should be portable, scaleable and simple for ease of implementing it on different machine and users.

The effectiveness of generic benchmarks not only depends on the above mentioned properties, but also on their wide acceptability and use by the system manufacturers. For this purpose these benchmarks need to be standardized. There are several organizations which are responsible for defining domain specific benchmarks, standard metrics, and standard format of measuring and reporting results. In the domain of software engineering the Transaction Processing Performance Council (TPC) defines benchmarks for transaction-processing and database domains. These benchmarks capture the complex query, batch, and operational aspects of transaction-processing systems. The TPC benchmarks are especially effective since the testing procedure, system price measurement, and formatting of results are well defined.

We have used the benchmarks to test the performance of both GDM and relational system (Ingres). The benchmark includes queries to test the performance of major components of a relational database system. It also emphasizes understanding the underlying semantics and statistics of the relations. This provides a much easier framework for adding new queries to better understand the behavior of the system to these benchmarks.

The results are summarized in Table 7.7. Based on these results we are of the opinion that GDM generally performs better than the relational system. Also it is amenable to implementation of an object oriented system.

	GDM	**Relation**
E/R Mapping	Excellent	Clumsy
Non-Recursive Queries	Fair	Excellent
Recursive Queries	Excellent	Fair
Object Orientation	Excellent	Excellent
Support of Language:		
Structural	Good	Good
Procedural	Good	No Support
Loading	Good	Fair

Table 7.7 Performance Comparison of GDM and Relational Models

7.7 Conclusion

In this chapter we have proposed a unique framework for managing large software projects using an object-oriented approach. Our approach is based on temporal modeling of software metrics data. Temporal modeling allows users to express complex scenarios spanning various metrics and helps in identifying risks and in evaluating the quality of the software project at various levels of abstraction.

The proposed approach uses a Petri net representation of temporal scenarios along with an iconic-based based representation of conventional database functionalities. Such a visual framework can provide a simple but comprehensive methodology expressing complex queries.