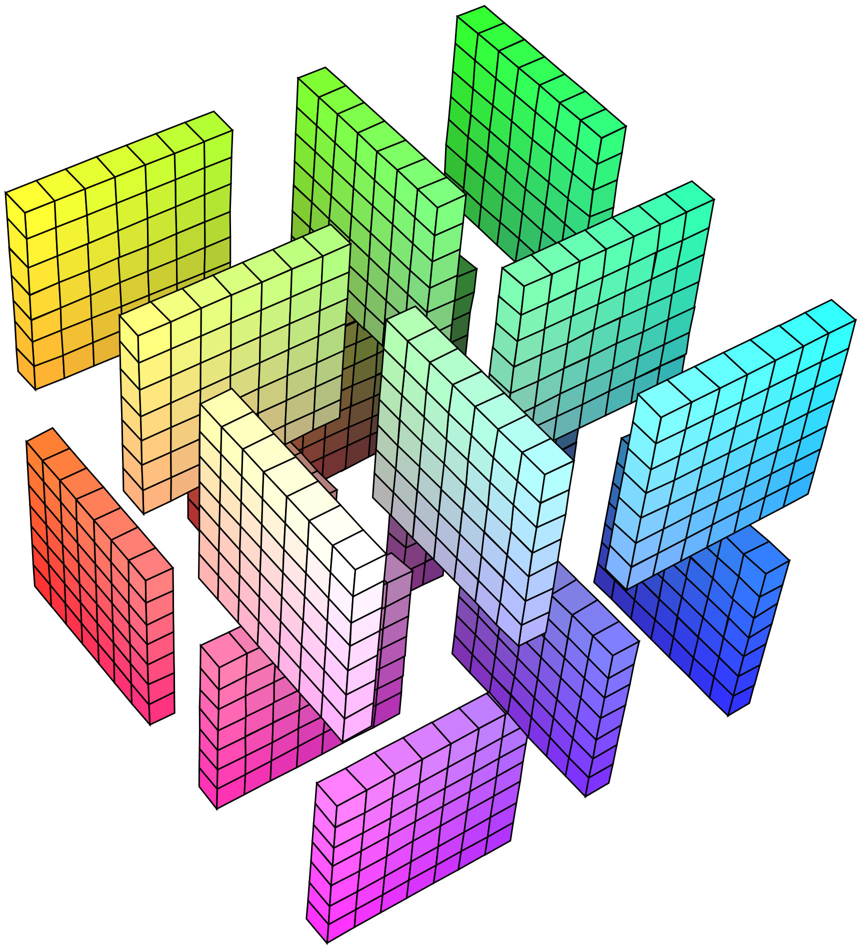
Linköping Studies in Science and Technology
Dissertation No. 452

ON EXTENSIBLE AND OBJECT-RELATIONAL DATABASE TECHNOLOGY FOR FINITE ELEMENT ANALYSIS APPLICATIONS

KJELL ORSBORN



Department of Computer and Information Science
Linköping University, Linköping, Sweden

Linköping 1996

Linköping Studies in Science and Technology
Dissertation No. 452

ON EXTENSIBLE AND OBJECT-RELATIONAL DATABASE TECHNOLOGY FOR FINITE ELEMENT ANALYSIS APPLICATIONS

KJELL ORSBORN

Department of Computer and Information Science
Linköping University, S-581 83 Linköping, Sweden

Linköping 1996

Cover illustration: "Color Slices" – from the Chromo Cube series – BECK & JUNG, 1996.
Used by permission of BECK & JUNG, Lund, Sweden.

ISBN 91-7871-827-9
ISSN 0345-7524

Printed in Sweden by triva-tryck ab, Linköping 1996.

PREFACE

The research presented in this thesis has been carried out at the Laboratory for Engineering Databases and Systems, Department of Computer and Information Science, Linköping University. It has been financially supported by the Swedish National Board for Industrial and Technical Development.

I would like to take this occasion to express my appreciation to my supervisor Professor Tore Risch for giving me the opportunity to carry out this research, and for his inspiring and excellent supervision during this work. Likewise, the additional supervision and encouragement from Associate Professor Bo Torstenfelt have been of invaluable importance. I would further like to direct my appreciation to Professor Sture Hägglund for his encouraging guidance during my research and to Associate Professor Anders Törne for his support during the initiation of this work. Furthermore, the generosity among my colleagues to share their knowledge and experiences, have greatly facilitated this work.

I would also like to acknowledge the artists BECK & JUNG for their kind generosity in giving me the permission to use one of their computer artwork in this thesis.

Linköping, August, 1996.

Kjell Orsborn

ABSTRACT

Future *database technology* must be able to meet the requirements of scientific and engineering applications. Efficient data management is becoming a strategic issue in both industrial and research activities. Compared to traditional administrative database applications, emerging scientific and engineering database applications usually involve models of higher complexity that call for extensions of existing database technology. The present thesis investigates the potential benefits of, and the requirements on, *computational database technology*, i.e. database technology to support applications that involve complex models and analysis methods in combination with high requirements on computational efficiency.

More specifically, database technology is used to model *finite element analysis (FEA)* within the field of *computational mechanics*. FEA is a general numerical method for solving partial differential equations and is a demanding representative for these new database applications that usually involve a high volume of complex data exposed to complex algorithms that require high execution efficiency. Furthermore, we work with *extensible* and *object-relational (OR)* database technology. OR database technology is an integration of *object-oriented (OO)* and *relational* database technology that combines OO modelling capabilities with extensible query language facilities. The term OR presumes the existence of an *OR query language*, i.e. a relationally complete query language with OO capabilities. Furthermore, it is expected that the *database management system (DBMS)* can treat extensibility at both the query and storage management level.

The extensible technology allows the design of *domain models*, that is database representations of concepts, relationships, and operators extracted from the application domain. Furthermore, the extensible storage manager allows efficient implementation of FEA-specific data structures (e.g. matrix packages), within the DBMS itself that can be made transparently available in the query language.

The discussions in the thesis are based on an initial implementation of a system called FEAMOS, which is an integration of a *main-memory resident* OR DBMS, AMOS, and an existing FEA program, TRINITAS. The FEAMOS architecture is presented where the FEA application is equipped with a local embedded DBMS linked together with the application. By this approach the application internally gains access to general database capabilities, tightly coupled to the application itself, that include a storage manager, a data model, a database language and processor, transactions, and remote access to data sources. On the external level, this approach supports concurrency, inter-operability, data exchange and transformation, data and operator sharing, data distribution, etc., among subsystems in an *engineering information system* environment. In the FEAMOS system, data representations and their related operators in TRINITAS have piece by piece been replaced by corresponding representations in AMOS. To be able to express matrix operations efficiently, AMOS has been extended with data representations and operations for numerical linear matrix algebra that handles overloaded and multi-directional foreign functions.

Performance measures and comparisons between the original TRINITAS system and the integrated FEAMOS system show that the integrated system can provide competitive performance. The added DBMS functionality can be supplied without any major performance loss. In fact, for certain conditions the integrated system outperforms the original system and in general the DBMS provides better scaling performance. It is the authors opinion that the suggested approach can provide a competitive alternative for developing future FEA applications.

CONTENTS

1 INTRODUCTION

Future *database management systems (DBMSs)* must be able to meet the requirements of scientific and engineering applications. Scientific and engineering data management is becoming a strategic issue in both industrial and scientific communities. A high leverage is confined in providing efficient information management and flexible information systems in enterprises as well as for research. In the engineering field, an *engineering information system (EIS)* is responsible for providing information among several engineering and business disciplines, as indicated in Figure 1, to support the complete product life-cycle of various products. Most simplified, the scientific field commonly has the problem of handling large amounts of empirical data sets provided by some test equipment on ground or in space. It is believed that database technology can play a similar and important role in the implementation of scientific and engineering applications of tomorrow, as it is currently doing in administrative applications.

In contrast to traditional administrative database applications, applications in science and engineering usually involve more complex models that need to be represented in the database. This calls for extensions of existing database technology to be able to handle these models efficiently [1] [2] [3].

Furthermore, there are activities concerned with various types of advanced analyses that include computational intensive tasks and form a subset of all activities that should be supported in a scientific or engineering information system. This can include several kinds of mechanical, electrical, chemical analyses, etc. These kinds of activities are also found in other fields such as in advanced financial and statistical applications. In addition to models of higher complexity, these activities include computational-intensive and complex analysis methods. Together, this requires that extensions of existing database technology should support data and operator representation capabilities that preserve efficient data processing. We use the term *computational database technology* to refer to database technology that should support applications emphasising processing efficiency and needs for complex and application-specific operations. It is intended that this should be a unifying term for database technology, in engineering, science, statistics, etc., that emphasise the computational aspect in addition to more conventional data management.

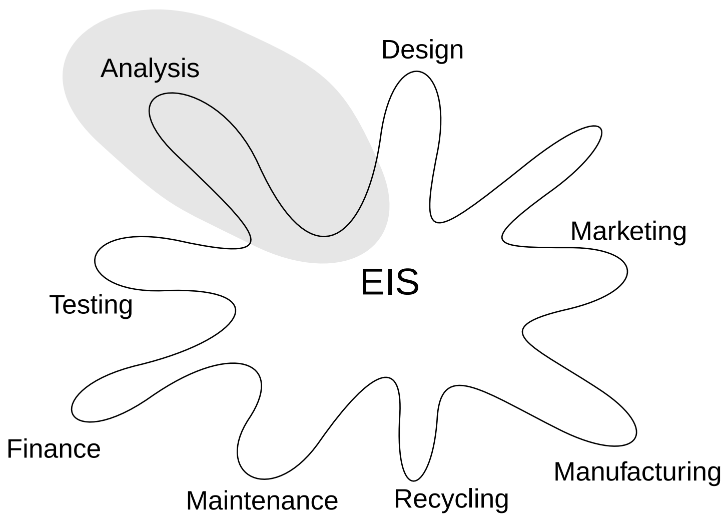


Figure 1. *Engineering information management should support several disciplines in an engineering information system environment.*

1.1 DATABASE TECHNOLOGY FOR FINITE ELEMENT ANALYSIS

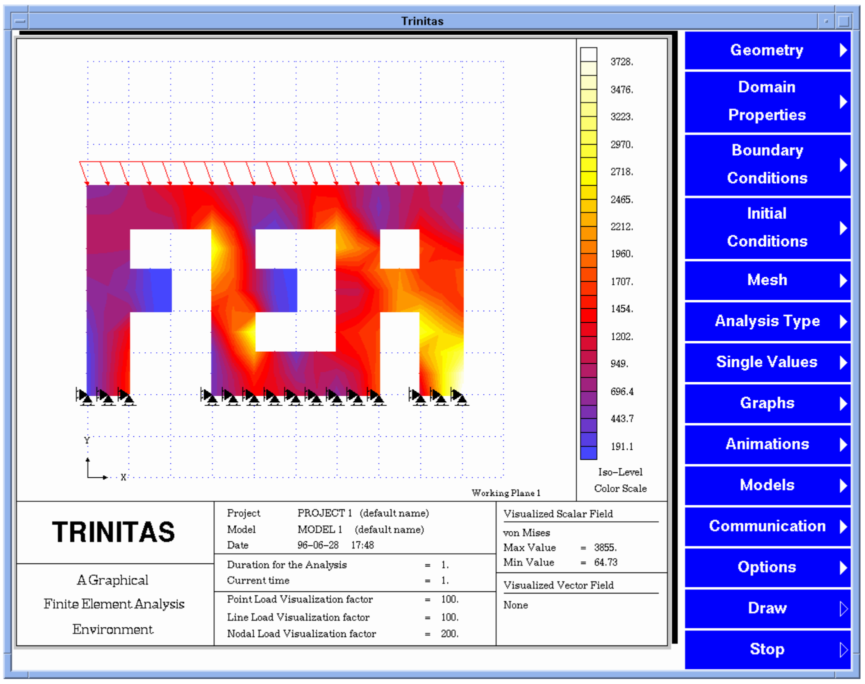
The present thesis focuses on database technology for applications within the computational mechanics field. The potential benefits of, and the requirements on, database technology for supporting these applications are investigated. More specifically, our work is on the *next generation extensible and object-oriented (OO) database technology*, also referred to as *object-relational (OR)* database technology, DBMS [4], Frank [5], and Stonebraker and Moore [6]. OR database technology is an integration of OO and relational database technology that combine OO modelling capabilities with query language facilities. Hence, OR presumes the existence of a *relationally complete OO query* *language*. Further it is expected that the DBMS can treat extensibility at both the query and storage management level. We use DBMS technology to model the field of *finite element analysis (FEA)*, a general numerical method for solving partial differential equations. FEA is a demanding representative of these new database applications that usually involve a high level of complexity of both data and algorithms, as well as a high volume of data and high requirements on execution efficiency. The discussions in the thesis are based on an initial implementation of a system called FEAMOS [7], which is an integration of a *main-memory (MM) resident* OR DBMS, AMOS [8] [9], and an existing FEA program, TRINITAS [10] [11].

The AMOS design intends to provide a lightweight and open DBMS architecture that should permit an easy combination and integration with other applications. It should further facilitate tailoring and extension of the DBMS to suit the needs of demanding applications as found in the engineering area. AMOS is intended to perform as a mediating software layer, [12] [13] [14], among applications and data sources for locating, storing, retrieving, exchanging, transforming, and monitoring data. AMOS can be an embedded database within an application by directly linking AMOS to the application at compile time. The application and the DBMS will then be executing in the same computer process and be sharing its address space. In addition, AMOS can be used in a conventional client-server environment where the applications and the DBMS have their own computer processes via the client-server interface. It is also possible to define domain-specific packages of specialised data structures and operators, and integrate them with AMOS. AMOS has the ability to seamlessly define and call foreign functions (implemented in C or LISP) through its foreign data source interface.

AMOS further includes the AMOSQL query language that, in this work, has been used to represent and manipulate the *domain conceptualisation*, i.e. concepts, relationships, and operations, of the FEA domain. AMOSQL is a more than relationally complete and extensible OO query language that is an extended derivative of OSQL, Lyngbaek [15]. The query language is influenced and has much in common with new standardisation efforts for query languages like SQL3, Melton [16], and OQL, Cattell [17].

TRINITAS is a general-purpose FEA program that integrates the entire analysis process and that can be completely controlled through a graphical user interface, illustrated in Figure 2. A typical TRINITAS session includes an interactive problem specification in terms of geometry, boundary conditions and domain properties. This is followed by a discretisation phase, a solution phase, and an evaluation of the results of the calculation. The TRINITAS system currently includes functionality for analysing static, dynamic, and eigenvalue problems within the mechanical design domain, including elastic and thermal effects. In addition, TRINITAS includes capabilities to handle adaptivity, optimization, and contact problems in static cases. The TRINITAS program does not incorporate any data or result files. Instead, all model interaction is performed through the graphical user interface that accesses main-memory data structures representing the analysis model. It is further designed in a highly structured, “object-based”, manner with specific sets of procedures for each concept, such as point or line.

In FEAMOS, both structure and process of the FEA domain are modelled in the database. This is done by defining a *domain model* using the extensible OR query language, the extensible query optimizer, and the extensible storage manager. A domain model represents a specific category of the mediator layers that are responsible for managing application-specific knowledge. The domain model is a database representation of concepts, relationships, and operators extracted from the application domain. In our case, a database schema is defined to represent finite element (FE) methodology, i.e. FE models and solution algorithms. The extensible query language allows domain-specific FE operators to be included in the DBMS. A user can define queries in terms of the FE model, and the queries may contain FE specific operators. By providing cost hints to the extensible query optimizer the execution cost of new operators in the query language can be treated by the optimizer. Furthermore, the extensible storage manager allows efficient implementation of FEA-specific data structures (e.g. matrix packages) within the DBMS itself and then made transparently available in the query language.



Project	PROJECT 1 (default name)		
Model	MODEL 1 (default name)		
Date	96–06–28 17:48		
Duration for the Analysis		=	1.
Current time		=	1.
Point Load Visualization factor		=	100.
Line Load Visualization factor		=	100.
Nodal Load Visualization factor		=	200.

Visualized Scalar Field

von Mises
Max Value = 3855.
Min Value = 64.73

Visualized Vector Field

None

Figure 2. *A “FEA model”, analysed in the FEAMOS system, shows a view of the graphical user interface of TRINITAS.*

The thesis presents an architecture for the integrated FEAMOS system where the FEA application is equipped with a local embedded MM DBMS that is linked into the application [18]. By this approach the application gains access to general database capabilities tightly coupled to the application itself, providing a storage manager, data model, database schema, database language and processor, transaction processing, and remote access to data sources. On the external level, this approach supports, for example, concurrency, inter-operability, data exchange and transformation, data and operator sharing, data distribution among applications and data sources in the engineering information system (EIS) environment. Different AMOS mediators are here responsible for locating, translating, and integrating data in various data sources for the applications. Ultimately, the DBMS can decide how and where to execute a query, using query optimization techniques. Internally, the architecture provides the application with powerful and high-level modelling capabilities through the object-relational query language. This includes object identities, subtyping, inheritance, views, overloaded functions, multi-directional functions, and foreign functions. The modelling capabilities make it possible to design database schemas that possess both physical and logical data and operator independence. Hence, the query language modelling supports and facilitates high-level application modelling that increases flexibility, composability, and reusability of domain conceptualisations.

It is of vital importance for the application to preserve the execution efficiency while adding functionality to the system. The present approach supports this requirement in several ways. Most important is the ability to provide an embedded database where the application can access and update data through a fast-path interface using precompiled and preoptimized database functions. The AMOS extensibility with foreign data sources, i.e. packages of specialised data representations and operations, makes it possible to provide efficiency for critical activities. For instance, scientific and engineering applications usually involve large amounts of numerical data that must be represented and processed effectively. By providing specialised representations and operations it is possible to avoid unnecessary copying and transformation of data. Execution efficiency is also supported by the query processor that has the ability to optimize access paths and operator ordering. This is especially important in complex modelling situations where the optimizer can automatically chose a good execution order. This simplifies the design of the database and frees the programmer from specifying the exact execution order which can be stated in higher-level terms. By providing general and efficient data representations in the DBMS, these become directly available to the application and need not be re-implemented.

In the FEAMOS system, data representations and their related operators in TRINITAS have piece by piece been replaced by corresponding representations in AMOS. All data, formerly residing in TRINITAS, is now stored in the database. The thesis show examples of how the query language can meet data modelling needs in different parts of the FEA domain including geometry, mesh, analysis algorithm, and calculated results. AMOS has also been extended with a foreign data source; a package for numerical linear matrix algebra. This package includes dense and skyline matrix representations integrated in a matrix-type structure in the database schema. The schema further includes functions for solving linear equation systems and everything is transparently integrated in the query language. To be able to express the matrix operations efficiently, AMOS has been extended to handle overloaded and multi-directional foreign functions [19], i.e. the ability to handle overloading of foreign functions on all arguments and for different binding patterns. All matrix representations are implemented by means of a basic data source for numerical arrays.

1.2 RESEARCH METHOD

The requirements and potential benefits of database technology for FEA applications have been studied and investigated primarily by implementing and testing database technologies for a real FEA application where the applicability can be evaluated and demonstrated. A fundamental issue is the ability to make changes and additions to, and replace source code in both the DBMS and in the FEA application. An iterative approach is used where parts of the application can be studied and where initial implementations are refined until a satisfactory result is achieved or other conclusions can be made.

The work presented in this thesis spans the fields of database technology and FEA technology. It has been an aim to put appropriate emphasis on both fields in order to avoid naive research contributions with respect to each field. The attempt to cover both fields, has meant that some losses in depth have probably been made in the treatment of specific parts in each area. Hopefully, this is more related to the nature of interdisciplinary research than to lack of insight on the part of the author.

1.3 RESEARCH SCOPE

This work has mainly covered database technology for FEA applications within the computational mechanics field. The emphasis has been on the local perspective in studying the representation and processing of FEA conceptualisations using query language technologies; in other words, how database technology can be used within an FEA application to support modelling and manipulation of FEA data. However, an important reason to include database facilities locally, within the application, is that this provides the application with mechanisms for communicating and exchanging data and information with other applications. Hence, the global perspective has also been considered in this work, which is revealed in the architectural discussions for FEA applications and EIS environments in general. Other issues are more related to the global perspective, such as distribution, replication, and concurrency. Further, transactional control of FEA activities has not been treated here but the potential benefits of transactions have been pointed out for future work.

Furthermore, the software-related issues of FEA have been emphasised and a restriction has been made to work with one specific FEA application. It has further been an aim to cover, at least to some extent, the various subactivities of a complete FEA to investigate the potential advantages that database technology can provide. The conceptualisation of the FEA domain has mainly been restricted to two-dimensional, static, and linear-elastic analyses, in order to treat a more manageable problem domain.

1.4 THESIS OUTLINE

After this introduction to the ideas behind this thesis, its outline will be briefly reviewed. The next two chapters, Chapter 2 and Chapter 3, continue with a presentation of FEA and database technology, the two major research fields of concern in this research. Chapter 2 starts by providing an intuitive introduction to the concepts of FEA and the process of carrying out an FEA. This is followed by a more mathematical derivation of the FEA concepts, within the scope of two-dimensional linear elasticity, to reveal the origin of various FEA quantities. Next, we turn to the description of conventional FEA software and points out some of their problems. Eventually, the background on FEA concludes by presenting the TRINITAS FEA application which has acted as the FEA software basis in this research. It should be noted that subsections are mainly directed to the potential reader who has little or no experience in FEA, where the last of these sections requires some insight into mathematical calculus. Readers from the FEA community can probably skip these parts. The rest of this chapter is intended for a broader audience.

The second field, database technology, is reviewed in Chapter 3. It starts with a review of the basic concepts and objectives of DBMSs. This is followed by a brief presentation of different categories of conventional database technology including: hierarchical, network, and relational database technology, that are categories mainly based on a division of database technology with respect to the underlying data model. The next sections presents database categories more relevant for this research, namely object-based, extensible, and main-memory database technology. The following section presents two additional categories, distributed and active DBMSs, that are included mostly for their potential long-term importance for this work. Section 3.7 discusses the characteristics and requirements of scientific and engineering database technology and specifically for FEA applications. Finally, this database technology chapter ends with a brief review of query languages for DBMSs. In similarity with the previous section on FEA technology, the two initial parts in this section are primarily directed to the reader with little or no knowledge in database technology.

This review of background information for this research is followed by a presentation, in Chapter 4, of AMOS and its basic technology, i.e. the specific DBMS that acts as the research tool in database technology within this research. It describes the mediator idea, the AMOS architecture, the AMOSQL language, and the facilities for embedding, interfacing, and extending AMOS.

The main chapter in this thesis, Section 5, treats the FEAMOS approach of using database technology for FEA applications and prototype development of an FEA application based on main-memory, extensible and OR database technology. This section starts by explaining and motivating the application of this kind of database technology to FEA and then continues to describe the architecture of FEAMOS. Thereafter follows a section that describes the extension of AMOS with numerical linear matrix algebraic capabilities using a matrix foreign data source. This is accomplished by the use of an array foreign data source that is also described. Next, the higher-level FEA domain modelling is treated with examples in representing geometry, mesh, algorithms, and results. The last section addresses performance issues by a few comparisons between FEAMOS and TRINITAS.

Before the summary, in Chapter 7, that discusses the FEAMOS approach and presents the conclusions, a short chapter, Chapter 6, provides some comments on alternative and related technologies. This includes both implementation techniques such as OO programming languages, relational database technology, OO database technology, and knowledge-based techniques, as well as standards for representing product data.

1.5 NOTATIONS

This section provides a short list of common notations used in the rest of this thesis. The list is as follows:

$\mathbf{A}$, σ, ε:	matrix
$\mathbf{A}_{square}$:	matrix subtype
$\mathbf{A}_{col}$, $\mathbf{A}_{row}$, $\mathbf{a}$:	column or row matrix
a_{ij}, a_x:	matrix components
A, a:	scalar
$\mathbf{A}^e$:	element quantity
$\mathbf{A}^T$:	transposed matrix

2 FINITE ELEMENT ANALYSIS AND SOFTWARE

The present chapter starts with an intuitive introduction to *finite element analysis (FEA)* followed by an outline of the FEA process. This is followed by a more formal presentation of the concepts of FEA by means of a specific example. These parts are mainly directed to readers with little or no experience of the FEA field and present the field in a form that should be relatively easy to penetrate. To a great extent, the notation follows the one found in Ottosen and Petersson [20]. The next parts are more directed to a general audience and include a description and discussion of software for FEA and the software environment in which it should be used. These parts further present current research directions in designing FEA software. Finally, this chapter ends with a presentation of TRINITAS, Torstenfelt et al. [10] and Torstenfelt [11], a state-of-the-art FEA research system that has formed the application base in this work.

2.1 FINITE ELEMENT ANALYSIS

FEA represents a broad class of approximate numerical analysis techniques to solve partial differential equations. Several scientific and engineering disciplines take advantage of these kinds of general analysis techniques. In the engineering field different classes of the *finite element method (FEM)* is applied to solve corresponding problems in areas such as electrostatics, electromagnetics, heat conduction, fluid flow, stress and strain, vibration, and stability [20] [21] [22]. The present treatise is biased towards, but not restricted to, the mechanical engineering field where FEA is used for analyses of designs involving different characteristic design criteria, such as strength, stiffness, stability, and resonance.

The application of FEM in an analysis situation could be intuitively described by means of a simple example, shown in Figure 3. The left part of Figure 3 represents a hypothetical problem where a steel console is rigidly fastened at the lower edge and is further exposed to a uniformly distributed traction load at the upper edge. For example, to be able to calculate the deformation and the corresponding internal loadings of the console, the analyst transforms this "real" problem into a corresponding FEA problem, here illustrated in the right part of Figure 3.

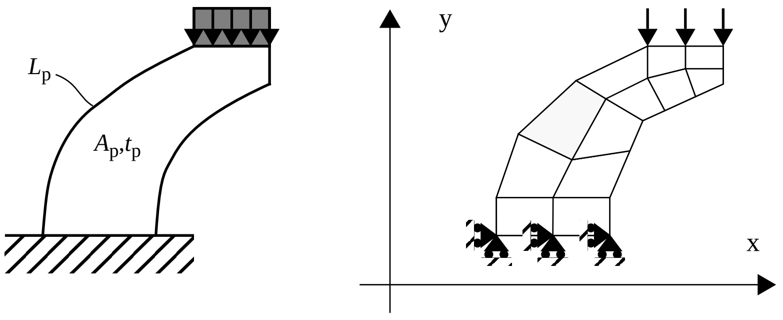


Figure 3. *The left part of the figure illustrates a rigidly fastened console exposed to a uniform traction at the upper edge and it is supposed it can be represented as a plane solid mechanical problem. It consists of a region, A_p, with a thickness, t_p. Further, the region is bounded in the plane by its boundary L_p. In the right part of the figure, a corresponding and schematic finite element model is presented. The FE-model consists of eight two-dimensional and linear finite elements that have rigid boundary conditions at the lower edge and nodal loads acting at the upper edge.*

The idea behind is to approximate the physical and continuous quantities of the "real" problem, such as shape, material, and loadings, with a corresponding set of piece-wise continuous quantities where the mathematical treatment should preserve important physical behaviour. This is accomplished by selecting and applying a set of predefined approximation functions for each quantity. For instance, the geometry is approximated with a set of finite elements, connected together at the corners that are also called nodes. Using the node coordinates, the geometry can be interpolated along element edges and within elements. In our example, the geometry is approximated by eight bilinear elements that form a piece-wise linear region. Similarly, other quantities can be approximated using interpolation functions. Usually, the displacement field is approximated in terms of the node displacements and will become the primary unknowns in the final equation system. Likewise, the rigid boundary condition will be expressed in terms of node displacements and the distributed load will be transformed into nodal loads.

Hence, this approximation technique transforms the continuous problem into a corresponding discrete problem that results in an equation system that in our example will be expressed in terms of the node displacements. The node displacements are then calculated by solving the equation system using numerical analysis techniques. Finally, the stress distribution in the body can be calculated from the displacements.

2.2 THE FINITE ELEMENT ANALYSIS PROCESS

The FEA process can typically be divided into four major activities *specification*, *discretisation*, *analysis*, and *evaluation*, as illustrated in Figure 4. First one needs to specify the problem with data about the shape, and about the *domain* and *boundary conditions*. The shape, or the geometry, is defined in terms of geometrical entities. Domain data that defines the material and boundary data can, for instance, consist of forces and prescribed displacements. Secondly, the discretisation activity decomposes the continuous domain into a finite element *mesh* consisting of *elements* and *nodes*. The mesh data is used in the subsequent analysis activity along with the domain and boundary conditions, to build the equation system to be solved. In a linear-elastic static analysis, this implies the solution of a single linear equation system expressed in matrix form, $\mathbf{K}\,\mathbf{a} = \mathbf{f}$, where $\mathbf{K}$ is the *stiffness matrix*, $\mathbf{a}$ the *displacement vector*, and $\mathbf{f}$ the *load vector*. The $\mathbf{K}$ matrix is assembled by stiffness contributions reduced to the nodes from every element and $\mathbf{f}$ includes load components from the boundary conditions reduced to appropriate nodes. The $\mathbf{a}$ vector represents the unknowns to be solved, i.e. displacements, and sometimes rotations, for every degree of freedom at the nodes, and is calculated during the analysis activity. The fourth activity, evaluation, includes result evaluation and validation of different levels of complexity. For instance, it might include a calculation, visualisation, verification, and an evaluation of critical quantities, such as stress fields, or deformation. The engineer usually decides which quantities should be investigated and further carries out the evaluation manually or semi-manually by means of computer support, e.g. the engineer can visually investigate computer-visualised stress fields of the model in search for critical areas that might imply that a redesign is necessary.

The evaluation might show that the specified requirements are met or it might indicate that a further and more detailed analysis must be considered or that a redesign must take place that again implies a re-analysis. This is indicated in Figure 4 where the whole process cycle might be looped several times until satisfactory results are accomplished. Likewise, different reasons may also imply iterations in the subactivities. Obviously, one might want to alter the geometry or the mesh before performing the analysis step. Furthermore, the analysis activity is sometimes repeated for different load cases. More complex analysis algorithms, such as algorithms for non-linear or dynamic problems, are iterative in themselves and they can further require an adjustment of the analysis parameters between analysis steps. If an analysis quantity must be evaluated in more detail, or if complementary results must be checked, the evaluation activity can also involve iteration before completion.

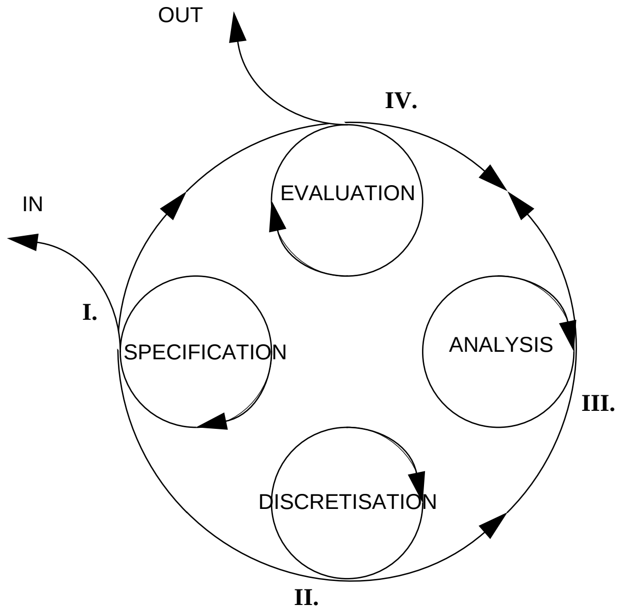


Figure 4. *The FEA process divided into four activities: I) problem specification in terms of geometry, boundary conditions and domain properties, II) discretisation (meshing) of analysis geometry into an approximate discrete representation (the FEA mesh), III) assemblage and analysis of the equation system, IV) evaluation and synthesis of calculated results.*

Besides these basic needs of iterations in the FEA process, more complex analysis classes also require iterations in this process, which is not indicated in Figure 4. For example, adaptive FEA methods use mesh refinement, and other techniques, for iteratively enhancing the solution accuracy. This means that an iteration involving the discretisation and the analysis activity is needed. Furthermore, optimization techniques, such as shape optimization, require a repetition of the complete FEA cycle until some specified stop criteria are fulfilled, since the analysis involves an alteration of the design geometry.

Thus, the central part in a FEA involves the solution of one or several systems of equations of different levels of complexity depending on the phenomenon studied. Even in the most basic analysis case this usually involves a large amount of data[1]. For example, the number of unknowns in the equation system can range from a hundred to several hundreds of thousands and beyond. This large set of data further has a high level of complexity since most concepts, including geometry, domain and boundary conditions, mesh, equations, and calculated results are related in some sense.

2.3 FINITE ELEMENT ANALYSIS CONCEPTS

When solving a problem by FEA, a specific formulation of the FEM is applied corresponding to that problem category. There exist numerous FEA formulations for different problem classes including boundary-value problems, initial-value problems, and eigenvalue problems. For instance, the static linear elasticity or heat conduction problems are formulated as elliptic partial differential equations that constitute subcategories of the boundary-value problem category [23].

In this context, we introduce the application of FEA by means of a class of problems restricted to plane linear-elastic static problems. This problem class is illustrated by Figure 5, where a plane body occupies region *A* and is restricted in the xy-plane by the boundary *L*. Furthermore, it is assumed that the interaction between the body and the environment can be stated as a combination of prescribed displacements on one part of the boundary, L_g, and of prescribed tractions on the other part of the boundary, L_h. Some further restrictions will be made in the subsequent presentation to facilitate the interpretation.

The governing equations for the mechanical problem of solids consist of the equations of equilibrium, the kinematic equations, and the constitutive equations. These equations are, together with appropriate boundary conditions, the basic equations of solid mechanics.

Considering the static case and ignoring body forces, the equations of equilibrium are given by

$$\tilde{\nabla}^T \sigma = 0 \qquad (1)$$

where σ is the stresses in the components

1. Data is in the FEA context used as a general term that refers to both input data and result data of an analysis.

$$\sigma = \begin{bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{xy} \end{bmatrix} \tag{2}$$

and $\tilde{\nabla}$ is a matrix differential operator in two dimensions defined as

$$\tilde{\nabla} = \begin{bmatrix} \frac{\partial}{\partial x} & 0 \\ 0 & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} \end{bmatrix} . \tag{3}$$

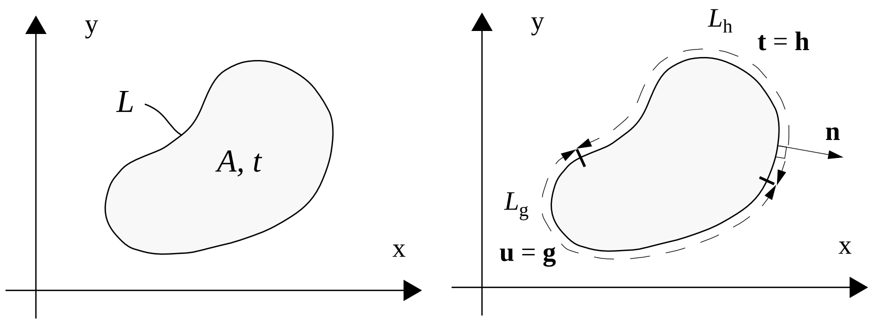


Figure 5. *The left part of the figure illustrates a general plane body that consists of a region, A, with a thickness, t. Further, the region is bounded in the plane by its boundary L with its normal vector **n**. In the right part of the figure the boundary has been divided into two parts L_g and L_h such that $L = L_g + L_h$. On L_g, the essential boundary condition **u** = g holds, whereas L_h is influenced by the natural boundary condition **t** = h.*

The kinematic relation defines the strains, ε, and states that

$$\varepsilon = \tilde{\nabla}\mathbf{u} \tag{4}$$

where **u** is the displacements that, in the two-dimensional case, has the components:

$$\mathbf{u} = \begin{bmatrix} u_x \\ u_y \end{bmatrix} . \tag{5}$$

The strain components in the two-dimensional case are

$$\varepsilon = \begin{bmatrix} \varepsilon_{xx} \\ \varepsilon_{yy} \\ \varepsilon_{xy} \end{bmatrix} . \tag{6}$$

If the thermal strains are excluded, the constitutive relation for linear elasticity, i.e. Hooke’s generalised law, states that:

$$\boldsymbol{\sigma} = \mathbf{D}\varepsilon \tag{7}$$

where **D** is the constitutive matrix. If we consider isotropic materials and plane stress conditions, **D** is given by

$$\mathbf{D} = \frac{\mathrm{E}}{1-\nu^2}\begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1}{2}(1-\nu) \end{bmatrix} \tag{8}$$

where E is Young’s modulus, and ν is Poisson’s ratio.

Boundary conditions can typically be expressed in terms of prescribed traction vectors, **t**, or displacements, **u**. In the two-dimensional case we have

$$\mathbf{t} = \mathbf{h} \qquad \text{on } L_\mathrm{h}\text{, and} \tag{9}$$

$$\mathbf{u} = \mathbf{g} \qquad \text{on } L_\mathrm{g} \tag{10}$$

where **h** is given on the L_h part of the boundary and **g** are given on the L_g part of the boundary. The type of boundary conditions represented by Eq. (9) are called *natural boundary conditions* since they follow from the statement of the problem whereas Eq. (10) represents boundary conditions that are called *essential boundary conditions*. As illustrated in Figure 5, the entire boundary L is the sum of L_h and L_g. Further, the traction vectors can be expressed as

$$\mathbf{t} = \mathbf{S}\,\mathbf{n} \tag{11}$$

where **S** is the stress tensor and **n** is the normal boundary vector. Their components are in two dimensions

$$\mathbf{t} = \begin{bmatrix} t_x \\ t_y \end{bmatrix}, \tag{12}$$

$$\mathbf{S} = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} \\ \sigma_{yx} & \sigma_{yy} \end{bmatrix}, \text{ and} \tag{13}$$

$$\mathbf{n} = \begin{bmatrix} n_x \\ n_y \end{bmatrix}. \tag{14}$$

The field equations, Eqs. (1), (4), and (7), are a general analytic formulation of the static and linear-elastic mechanical problem for isotropic solids.

A FEA formulation corresponding to this basic problem statement can be stated from a weak formulation of the equilibrium equations Eq. (1). This process includes the introduction of a vector-valued weight function and the application of the well-known Green-Gauss theorem, that results in a transformation of Eq. (1) to

$$\int_A (\tilde{\nabla}\mathbf{v})^T \sigma t dA = \oint_L \mathbf{v}^T \mathbf{t}\, t dL \tag{15}$$

where **v** is an arbitrary weight function. Further, according to Figure 5, *A* is the plane region of the body that is circumscribed by the boundary *L* and has the thickness *t*. The left-hand side of Eq. (15) represents the internal balance term that should be in balance with the boundary term represented by the right-hand side. The establishment of this equation only involves the equilibrium equation in Eq. (1) and, hence Eq. (15) is not restricted to any specific constitutive model.

From the weak formulation of the balance equation, Eq. (15), the FEA approximation can be introduced in a straightforward manner since the weak formulation only restricts the approximated quantities to be piece-wise continuous within the region *A*. This criterion is fulfilled by choosing the approximations according to

$$\mathbf{u} = \mathbf{Na} \tag{16}$$

where **N** contains the global interpolation functions and **a** contains the nodal displacements. The components of **N** and **a** are:

$$\mathbf{N} = \begin{bmatrix} N_1 & 0 & N_2 & 0 & N_3 & \dots & N_n & 0 \\ 0 & N_1 & 0 & N_2 & 0 & \dots & 0 & N_n \end{bmatrix}, \text{ and} \tag{17}$$

$$\mathbf{a}^T = \begin{bmatrix} u_{1x} & u_{1y} & u_{2x} & u_{2y} & \dots & u_{nx} & u_{ny} \end{bmatrix}. \qquad (18)$$

In accordance with the Galerkin weighted residual method, the arbitrary weight function **v** should take the same approximation as **u** that yields

$$\mathbf{v} = \mathbf{N}\mathbf{c} \qquad (19)$$

where **c** is arbitrary.

Introducing **B** as

$$\mathbf{B} = \tilde{\nabla}\mathbf{N} \qquad (20)$$

and inserting Eqs. (19) and (20) in Eq. (15) yields

$$\mathbf{c}^T\left(\int_A \mathbf{B}^T\sigma t dA - \oint_L \mathbf{N}^T\mathbf{t}\ t dL\right) = 0. \qquad (21)$$

Since **c** is arbitrary it follows that

$$\int_A \mathbf{B}^T\sigma t dA - \oint_L \mathbf{N}^T\mathbf{t}\ t dL = \mathbf{0}. \qquad (22)$$

As for Eq. (15), this equation holds for arbitrary constitutive relations since we have not so far used any information about the material condition.

A constitutive model for linear elastic and isotropic materials, Eq. (7), is now introduced. The kinetic relation, Eq. (4), together with Eqs. (16) and (20) yield

$$\boldsymbol{\varepsilon} = \mathbf{B}\ \mathbf{a} \qquad (23)$$

and together with Eq. (7) we get

$$\sigma = \mathbf{D}\ \mathbf{B}\ \mathbf{a}. \qquad (24)$$

Insertion of Eq. (24) in Eq. (22) gives us

$$\left(\int_A \mathbf{B}^T\mathbf{D}\ \mathbf{B}\ t dA\right)\mathbf{a} = \oint_L \mathbf{N}^T\mathbf{t}\ t dL. \qquad (25)$$

This equation can be rewritten using the boundary conditions in Eq. (9). The complete boundary conditions are available since **t** is known along L_h and **u** along L_g. This yields:

$$\left(\int_A \mathbf{B}^T\mathbf{D}\,\mathbf{B}\; tdA\right)\mathbf{a} = \oint_{L_h} \mathbf{N}^T\mathbf{h}tdL + \oint_{L_g} \mathbf{N}^T\mathbf{t}\; tdL\,. \tag{26}$$

This equation is the finite element formulation of two-dimensional elasticity. Taking the definition of **D** for plane stress condition would result in a form that applies for plane stress and isotropic linear elasticity.

It is common to introduce the following notation to simplify the expression of Eq. (26):

$$\mathbf{K} = \int_A \mathbf{B}^T\mathbf{D}\,\mathbf{B}\; tdA\text{, and} \tag{27}$$

$$\mathbf{f} = \oint_{L_h} \mathbf{N}^T\mathbf{h}tdL + \oint_{L_g} \mathbf{N}^T\mathbf{t}\; tdL \tag{28}$$

where **K** is called the stiffness matrix and **f** the load vector. The **f** vector can include some additional terms that are not included since body forces and thermal strains are ignored. Furthermore, for the special case where L_g is fixed, i.e. **g** = 0, the second term in Eq. (28) vanishes and we get

$$\mathbf{f} = \oint_{L_h} \mathbf{N}^T\mathbf{h}tdL \tag{29}$$

With the notations of Eq. (27) and Eq. (29) we then have

$$\mathbf{K}\,\mathbf{a} = \mathbf{f}\,. \tag{30}$$

The equation Eq. (30) represents a linear equation system expressed in global quantities. The corresponding local form is straightforwardly accomplished by stating the equations in local quantities. Thus, at the element level we have

$$\mathbf{K}^e\mathbf{a}^e = \mathbf{f}^e \tag{31}$$

where

$$\mathbf{K}^e = \int_{A_\alpha} \mathbf{B}^{eT}\mathbf{D}\,\mathbf{B}^e tdA\text{, and} \tag{32}$$

$$\mathbf{f}^e = \oint_{L_{h\alpha}} \mathbf{N}^{eT}\mathbf{h}tdL\,. \tag{33}$$

The *e*, and α refer to the quantities of one element.

In more detail the quantities at the element level are commonly stated by means of an isoparametric formulation of the finite elements. An isoparametric element formulation provides elements that are allowed to be distorted more freely than simpler elements. This is accomplished by mappings from a parameterised parent domain to the global domain. We illustrate this for a four-node isoparametric element shown in Figure 6.

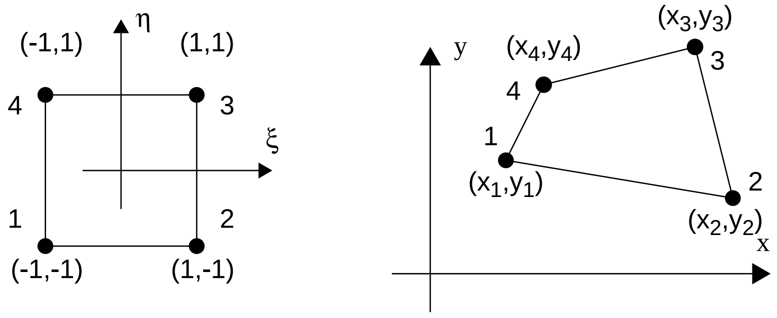


Figure 6. *The four-node isoparametric quadrilateral element. The parent domain (left figure) is expressed by the parameters ξ and η, both with the range (-1,1), that are used to express the mapping into the global domain (right figure).*

Hence, in two dimensions the coordinates x and y in the global domain are expressed by means of the parameters ξ and η in a parent domain. The mapping is performed by element interpolation functions for the corner points that in this case are:

$$\begin{aligned} N_1^e &= \frac{1}{4}(\xi - 1)(\eta - 1) \\ N_2^e &= -\frac{1}{4}(\xi + 1)(\eta - 1) \\ N_3^e &= \frac{1}{4}(\xi + 1)(\eta + 1) \\ N_4^e &= -\frac{1}{4}(\xi - 1)(\eta + 1) \end{aligned} \quad . \tag{34}$$

Using these interpolation functions the global coordinates x and y can be expressed as

$$x = x(\xi, \eta) = \mathbf{N}^e(\xi, \eta)\mathbf{x}^e \text{, and} \tag{35}$$

$$y = y(\xi, \eta) = \mathbf{N}^e(\xi, \eta)\mathbf{y}^e \tag{36}$$

where $\mathbf{N}^e$ for this four node elements is given by

$$\mathbf{N}^e(\xi, \eta) = \begin{bmatrix} N_1^e & N_2^e & N_3^e & N_4^e \end{bmatrix} \tag{37}$$

and where $\mathbf{x}^e$ and $\mathbf{y}^e$ have one component for each node that in this case results in

$$\mathbf{x}^{eT} = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \end{bmatrix} \tag{38}$$

for $\mathbf{x}^e$ and, likewise, for $\mathbf{y}^e$

$$\mathbf{y}^{eT} = \begin{bmatrix} y_1 & y_2 & y_3 & y_4 \end{bmatrix} . \tag{39}$$

Equations (35) and (36) can be used in expressions that include dependencies of x and y. However, to be able to evaluate the element quantities in Eqs. (32) and (33), they have to be transformed to the parent (ξ,η) domain. Performing these transformations yield the corresponding equations

$$\mathbf{K}^e = \int_{-1}^{1}\int_{-1}^{1} \mathbf{B}^{eT}(\xi, \eta)\mathbf{D}(\xi, \eta)\mathbf{B}^e(\xi, \eta)t(\xi, \eta)|\mathbf{J}|d\xi d\eta \tag{40}$$

where $\mathbf{B}^e$, the derivative of $\mathbf{N}^e$, is given by

$$\mathbf{B}^e(x, y) = \begin{bmatrix} \frac{\partial N_1^e}{\partial x} & 0 & \frac{\partial N_2^e}{\partial x} & 0 & \frac{\partial N_3^e}{\partial x} & 0 & \frac{\partial N_4^e}{\partial x} & 0 \\ 0 & \frac{\partial N_1^e}{\partial y} & 0 & \frac{\partial N_2^e}{\partial y} & 0 & \frac{\partial N_3^e}{\partial y} & 0 & \frac{\partial N_4^e}{\partial y} \\ \frac{\partial N_1^e}{\partial y} & \frac{\partial N_1^e}{\partial x} & \frac{\partial N_2^e}{\partial y} & \frac{\partial N_2^e}{\partial x} & \frac{\partial N_3^e}{\partial y} & \frac{\partial N_3^e}{\partial x} & \frac{\partial N_4^e}{\partial y} & \frac{\partial N_4^e}{\partial x} \end{bmatrix} \tag{41}$$

and where $|\mathbf{J}|$ is the Jacobian and is the determinant of the Jacobian matrix $\mathbf{J}$ that in two dimensions has the following form:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} \end{bmatrix} . \tag{42}$$

$\mathbf{J}$ is derived from the relation:

$$\begin{bmatrix} dx \\ dy \end{bmatrix} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} \end{bmatrix} \begin{bmatrix} d\xi \\ d\eta \end{bmatrix}. \tag{43}$$

Since the functions ξ(x,y) and η(x,y) are not normally known we can determine the inverse relation between the interpolation functions of the parent and the global domain in order to determine the partial derivatives in Eq. (41). Using Eq. (42) we get:

$$\begin{bmatrix} \frac{\partial N_i^e}{\partial \xi} \\ \frac{\partial N_i^e}{\partial \eta} \end{bmatrix} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} \end{bmatrix} \begin{bmatrix} \frac{\partial N_i^e}{\partial x} \\ \frac{\partial N_i^e}{\partial y} \end{bmatrix} = \mathbf{J}^T \begin{bmatrix} \frac{\partial N_i^e}{\partial x} \\ \frac{\partial N_i^e}{\partial y} \end{bmatrix} \tag{44}$$

Hence, if **J** is invertible we have

$$\begin{bmatrix} \frac{\partial N_i^e}{\partial x} \\ \frac{\partial N_i^e}{\partial y} \end{bmatrix} = (\mathbf{J}^T)^{-1} \begin{bmatrix} \frac{\partial N_i^e}{\partial \xi} \\ \frac{\partial N_i^e}{\partial \eta} \end{bmatrix} \tag{45}$$

that can be computed for each N_i.

Further, the element load vector $\mathbf{f}^e$ gets the following form

$$\begin{aligned} \mathbf{f}^e &= \oint_{L_{g\alpha}} \mathbf{N}^{eT} \mathbf{h}(x(\xi,\eta), y(\xi,\eta)) t(x(\xi,\eta), y(\xi,\eta)) dL \\ &+ \oint_{L_{g\alpha}} \mathbf{N}^{eT} \mathbf{t}(x(\xi,\eta), y(\xi,\eta)) t(x(\xi,\eta), y(\xi,\eta)) dL \end{aligned} \tag{46}$$

where

$$dL = \left[\left(\frac{\partial x}{\partial \xi} d\xi + \frac{\partial x}{\partial \eta} d\eta \right)^2 + \left(\frac{\partial y}{\partial \xi} d\xi + \frac{\partial y}{\partial \eta} d\eta \right)^2 \right]^{1/2}. \tag{47}$$

The boundary integrals in Eq. (46) must be evaluated for each of the four boundaries where ξ and η are equal to -1 and 1. For example, for ξ = ±1, Eq. (47) will take the form:

$$dL = \left[\left(\frac{\partial x}{\partial \xi}\right)^2 + \left(\frac{\partial y}{\partial \xi}\right)^2\right]^{1/2} |d\xi| \quad . \tag{48}$$

If we divide $\mathbf{f}^e$ into $\mathbf{f}^e_{L_h}$ on L_h and $\mathbf{f}^e_{L_g}$ on L_g and suppose that $\mathbf{h}$ is prescribed for the element edge where $\xi = 1$, the contribution to the load vector $\mathbf{f}^e$ would be

$$\mathbf{f}^e_{L_h} = \int_{-1}^{1} \mathbf{N}^{eT}\mathbf{h} \tag{49}$$

Likewise, to calculate the contribution to the load vector for loads on the other element edges we evaluate Eq. (46) for other values of ξ and η.

The central concepts of FEA have been introduced to show an example of what type of information should be represented within a FEA application. The FEA application requires further functionality to manage this information. As outlined in the previous section, the application needs numerical analysis capabilities to handle equation solving. A fully integrated FEA system would also include functionality to handle geometry, discretisation, and result evaluation, preferably supplied through a graphical user interface. The next section will continue the discussion on conventional FEA software and Chapter 5 will present how to take advantage of database technology for representing and managing FEA concepts.

2.4 SOFTWARE FOR FINITE ELEMENT ANALYSIS

FEA software is widely used in different engineering and scientific disciplines, where analysis of mechanical designs represents a main application area. A mechanical design can be analysed with respect to several phenomena, such as mechanical, thermal, and acoustic behaviour. Since the analysis requirements vary a great deal depending on the complexity of the design and its intended functionality, the software requirements of FEA programs vary in a similar manner. For example, for a simple design it might be sufficient with a single linear static analysis. This should be compared to design situations where large, complex and interrelated analyses of several analysis cases are performed that might further concern several parts of a design and include coupled phenomena.

This diverse complexity makes FEA strongly dependent on an efficient computing environment including both hardware and software. For the software area this not only includes the use of FEA programs but involves a much broader spectrum of software engineering issues. Firstly, considering internal issues while looking at an autonomous FEA program, it should ultimately be designed in a way that supports both effective usage as well as development and maintenance. Secondly there are inter-related aspects, where FEA software as any other EIS software ultimately should be designed to support effective integration and communication with other applications in an EIS software environment.

Modern commercial FEA programs have integrated the complete analysis process from modelling to evaluation and take advantage of graphical user interfaces where the analysis model can be specified in domain-specific terminology. However, when turning to FEA programs for more advanced analysis methods, it is not uncommon that several programs are involved in one analysis. A computer-aided design (CAD) program can be used to define the geometry, a second preprocessor program can be responsible for generating the mesh that should be supplied to the actual FEA program for the analysis activity. Finally, a postprocessor can be involved in visualisation and evaluation of analysis results. In this process data is typically exchanged through files of different formats.

Several among the major commercial FEA programs have their origin in the 1960's. In contrast to the exceptional development in hardware performance during the last 30 years, the basic structure of commercial FEA programs and their development have not gone through any dramatic change since their origin. This is partly explained by the fact that it is much easier to take advantage of hardware performance than to redesign and reimplement the software.

The conventional and direct use of FEA programs is mainly concerned with its analysis functionality, processing efficiency, and efficient user interfaces, and does not imply any direct requirements on its internal structure and flexibility. Efficient processing implies that efficient algorithms and corresponding data structures are available. The importance of flexibility and internal structure becomes more evident when turning to situations where data should be communicated to and from other systems, combined with other data, or composed into new derived information. The same holds for development and maintenance of this type of software where the complexity can be reduced and a higher level of reuse can be accomplished by increasing structure and composability.

In conventional FEA software, data and algorithms are usually integrated and designed for a specific purpose. Likewise, domain knowledge, such as consistency checks, are usually compiled into the application. To provide data exchange with other applications, specific interface programs must be written to access data. Further, the domain knowledge can not be inspected, verified, modified, or extended without writing a program. A more open software design where data and knowledge could be represented more explicitly would enhance the usability, maintainability, and the verification possibilities and probably increase the subsequent analysis quality.

Furthermore, investments in FEA and related software usually involve large direct costs as well as educational costs and potential costs for transformation of old data. Likewise, the software vendors have large investments in existing systems where a technology change in implementation technique becomes very costly. Consequently, these circumstances prohibit the evolution of FEA software.

If FEA data could be represented in a vendor-independent manner outside of FEA applications, a more flexible situation can arise. Hence, data should be modelled and accessed by as generic and standardized software tools and techniques as possible. Data modelling and management should rather be problem- and theory-dependent than application-program dependent. However, representing data independent of its usage might sometimes be impossible for efficiency reasons and must be considered in designing software tools and standards for EIS.

In order to increase the functionality and renew the design of scientific and engineering software in general, several modern programming techniques have been paid some attention, including knowledge-based techniques, Chalfan [24], Alsina et al. [25], Mitchell et al. [26], and Abelson et al. [27]; OO programming, Forde et al. [28]; and database techniques, Ahmed et al. [29], Eastman [30], Beck et al. [31], and Samaras et al. [32]. Likewise, the FEA research community, has applied knowledge-based techniques in several areas of FEA from supporting input data generation and mesh generation to the control of a complete analysis and to provide design knowledge, Mackerle and Orsborn [33], Forde and Stiemer [34], Ramirez and Belytschko [35], Shephard et al. [36], and Tworzydlo and Oden [37].

As in several other fields, OO programming languages, such as C++, CommonLisp (including CLOS), OO dialects of Pascal, and Smalltalk, have been suggested for design and implementation of FEA software, Baugh and Rehak [38], Fenves [39], Forde et al. [40], Filho and Devloo [41], Dubois-Pelerin et al. [42], Williams et al. [43], Scholz [44], Baugh and Rehak [45], Mackie [46], Ross et al. [47], Raphael and Krishnamoorthy [48], Yu and Adeli [49], Hoffmeister et al. [50], Arruda et al. [51], Devloo [52], Eyheramendy and Zimmermann [53], Gajewski [54], Ju and Hosain [55], Shepherd and Lefas [56], Langtangen [57], Cardona et al. [58], Zeglinski et al. [59], and Lu et al. [60]. A major reason for this has been to reduce program complexity by introducing OO structure in the software which is at least intuitively motivated since it is quite natural to think of engineering data in terms of objects and their relationships. A certain scepticism has sometimes been raised against these techniques directed towards a potential loss in execution efficiency. However, Devloo [52] shows that this is not the case, which is also supported in the DIFFPAK project, Langtangen [57], where even better performance compared to FORTRAN implementations has been reported.

The functional programming paradigm has also been suggested for implementing FEA software, Grant et al. [61].

In the FEA field, database support has so far been used for storage and retrieval of data and results mainly using relational databases and special-purpose database implementations, Yeh et al. [62], Felippa [63], Dopker et al. [64], Santana et al. [65], Myers [66], Xingjian [67], Spainhour et al. [68], Krishnamoorthy and Umesh [69], Pepper and Marino [70], Magnin and Coulomb [71], Yang and Yang [72], Baker [73], Felippa [74], and Bergman et al. [75]. It has further been shown in Ketabchi et al. [76] that OO DBMSs are more suitable than traditional DBMSs for modelling data in the engineering field.

Emerging standards for representing and exchanging product data will probably also play an important role in future engineering software. The STEP (STandard for the Exchange of Product data) standard covers the modelling of engineering data, ISO [77], and indeed FEA data, ISO [78]. The STEP standard is based on the data modelling language EXPRESS, ISO [79], that is used to specify data schemes for various engineering domains. Different tools to support EXPRESS-based data exchange are also being developed. However, these standards do not solve, and should probably not be considered as the final solution to, the complete management needs of engineering data. There will always be enterprise-specific data and use of data that does not conform to existing standards. For this reason, it ought to be convenient to combine or integrate standards like STEP with more general data management standards such as query languages for databases. For instance, SQL, the standard query language for *relational (R)* DBMSs, has a very broad coverage and is not restricted to any specific application area. A further extension of SQL to enable object-orientation is proposed in the SQL3 standard specifications [16]. Another competing standard proposal in this area is the ODMG standard[1], Cattell [17], that incorporates the OQL query language.

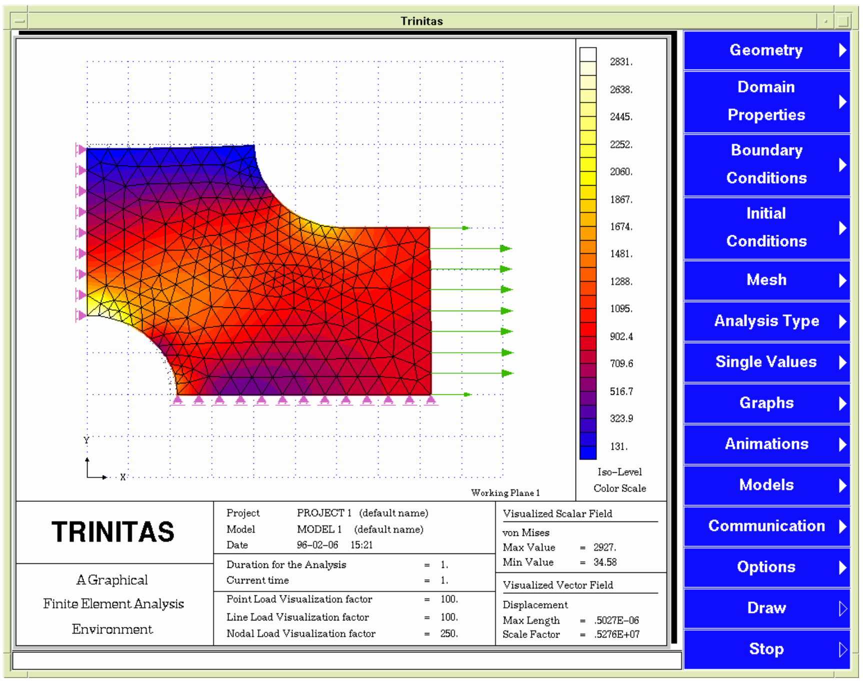
2.5 THE TRINITAS SOFTWARE

TRINITAS, Torstenfelt et al. [10] and Torstenfelt [11], is a general-purpose FEA program that integrates the entire analysis process and that is completely controlled through a graphical user interface, as illustrated in Figure 7. The typical TRINITAS session starts with an interactive problem specification in terms of geometry, domain properties and boundary conditions. An approximation of the geometry is then accomplished in the discretisation phase. The discretised geometry is thereafter used, in combination with boundary and domain conditions, to establish the equation system to be solved. This activity is an integrated part of the solution phase where the equation system also is solved. Eventually, the session ends with an evaluation of the results of the calculation. The TRINITAS system currently includes functionality for analysing static, dynamic, and eigenvalue problems within the mechanical design domain, including elastic and thermal effects. In addition, TRINITAS includes capabilities to handle adaptivity, optimization, and contact problems in static cases.

The TRINITAS program is “model-oriented” rather than file-oriented and does not incorporate any data or result files. Instead, all model interaction is performed through the graphical user interface that accesses main-memory data structures representing the analysis model. TRINITAS is further designed in a highly structured, “object-based”, manner with specific sets of procedures for each concept class, such as point, line, surface, and volume. The design is also layered with well-defined interfaces between the layers. A simplified view of the different layers is provided in Figure 8. These layers include the graphical user interface level, the application concepts layer including FEA-

1. The ODMG standard originates from the Object Management Group (OMG), Framingham, MA, USA.

related concepts and operations, the abstract array layer, and the data file layer for secondary storage. Additional interfaces exist for communicating with, for example, graphical devices. All data in the application layer is stored in main-memory using the abstract array representation and storage and retrieval to and from secondary storage is handled automatically by the system. The TRINITAS system currently consists of about 2200 subroutines of FORTRAN code and a small part of C code to interface graphics libraries.



```
Project        PROJECT 1   (default name)
Model          MODEL 1     (default name)
Date           96–02–06    15:21

Duration for the Analysis                =   1.
Current time                             =   1.

Point Load Visualization factor          =   100.
Line Load Visualization factor           =   100.
Nodal Load Visualization factor          =   250.

Visualized Scalar Field
von Mises
Max Value      =  2927.
Min Value      =  34.58

Visualized Vector Field
Displacement
Max Length     =  .5027E–06
Scale Factor   =  .5276E+07
```

Figure 7. *Example of the graphical user interface of TRINITAS.*

The geometry is the central concept when specifying an analysis model in TRINITAS. A analysis geometry is built up from basic geometric entities, such as points, lines, surfaces, and volumes. When the geometry is specified, it can be extended with different forms of boundary conditions, such as point loads, distributed loads, fixed or prescribed displacements. The domain properties are provided by default if nothing else is specified. Hence, the problem to be analysed can be completely specified before one considers how it should be discretised. This makes it possible to change the discretisation without respecifying the boundary conditions.

TRINITAS supports both mapped and free-mesh algorithms for the discretisation phase. About eight different element types are available, such as constant and linear strain triangles, bilinear quadrangle, quadratic lagrange, and trilinear hexahedral.

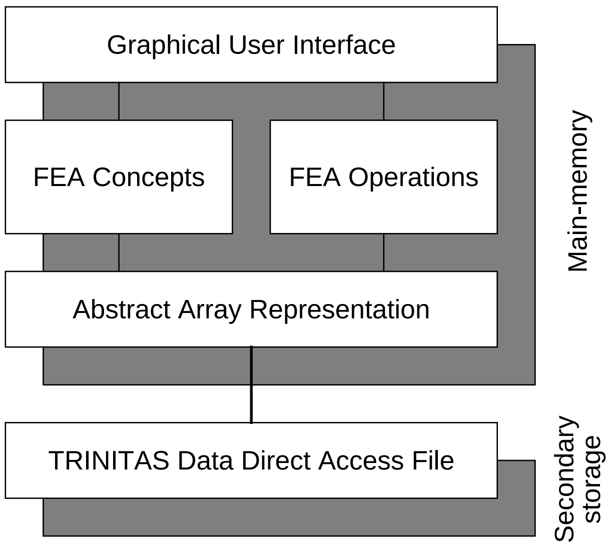


Figure 8. *A simplified view of the TRINITAS architecture.*

Information about relationships between geometric entities and the elements and nodes in the discretisation is used for calculating the load vector and to establish the stiffness matrix. Stiffness matrices can be stored either as full regular matrices or as compact skyline matrices. The analysis type automatically selects an appropriate representation. TRINITAS further includes an algorithm that performs bandwidth reduction before the equation solving takes place.

A number of analysis types are supported with an emphasis on linear elasticity. This includes conventional stress analysis as well as optimization of weight, stiffness, or stress, adaptivity, and contact analysis.

A variety of quantities from the analysis can be evaluated. These include displacements, basic stress components, von Mises stresses, and reaction forces. It is possible to evaluate both single values or to present a sequence of values in a graph. Furthermore, the system can automatically handle updates of these values if the analysis is recalculated. Field quantities can further be displayed as iso-level curves.

In Appendix A, a list of main concepts and analysis capabilities of TRINITAS is included. For further information on TRINITAS functionality, the reader is referred to [11].

As a final but important note, it is worth mentioning that the “object-based” and layered design of TRINITAS, illustrated in Figure 8, has facilitated the integration with the DBMS in this work. The TRINITAS architecture has made it possible to transfer subsets of the application model to corresponding database representations. In the FEAMOS system, the abstract array layer and the data file layer have been replaced by a corresponding database representation by introducing an interface to the database between the application layer and the abstract array layer. In addition, specific parts of the FEA-related concepts and operations could be separated and replaced by higher-level object representations within the database.

3 DATABASES AND DATABASE MANAGEMENT SYSTEMS

An effective operation of information assets is becoming a strategic issue in commercial activities. Formerly, these issues were mostly emphasized in administrative areas but have lately also got much attention in several engineering disciplines. The objective of the database management approach is to provide developers, administrators, and users with generic software tools that support definition and manipulation of data in an efficient, uniform, flexible, and secure manner.

A *database* is, according to Elmasri and Navathe [80], a collection of related data. Databases also usually incorporate further implicit properties in that the database represents a specific subdomain of the real world, the data is logically structured with an intended meaning, and the database is produced for a specific purpose. Elmasri and Navathe also define a *database management system* (DBMS) as a general-purpose software system that facilitates definition, construction, and manipulation of databases for various applications. As illustrated in Figure 9, a database and a database management system are together referred to as a *database system* (DBS) and might also include other application software.

A DBMS works as an intermediate layer between applications or users and data to provide a generic interface to the data. It should be viewed as a tool to protect data assets, improve data quality, and to facilitate changing informations needs according to

Loomis [81]. These generic software tools in the DBMS can help the developers, administrators, and users to define and manipulate data in a uniform manner.

A *database language (DBL)*, provided by a DBMS, is the actual interface for users and applications using a database. The DBL can be an integrated language that includes constructs for database definition and manipulation. Probably the most well-known integrated database language is SQL (Structured Query Language) [82], a standardized language for relational databases. The database community usually uses the term *query language* as a synonym for an integrated database language even though “query” refers only to retrieving data from that database. This habit is inherited by the present author.

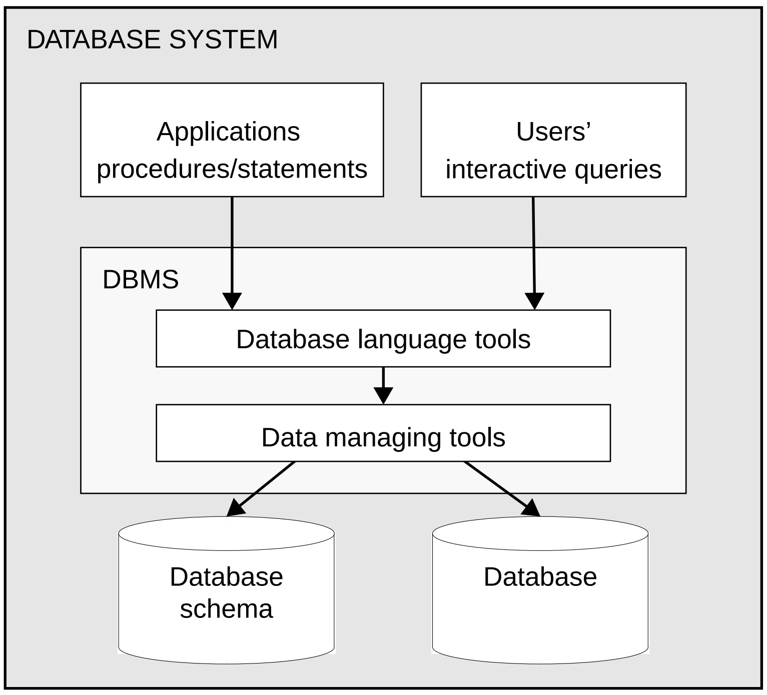


Figure 9. *Outline of a simplified database system.*

Before a database can be accessed, its content must be defined and it must further be populated with data. A database is defined by means of the DDL that includes constructs for defining a database *schema*, i.e. the structure of the database including data types, relationships, and constraints, and should reflect the structure of the application domain under consideration. A database schema is also referred to as the *system catalogue, data dictionary*, or *meta data*. This schema definition is made in terms of the *data* *model* that is supported by the DBMS. A data model is a set of predefined concepts including data types and basic operators provided by the DBMS. In addition, the behaviour of the application domain under consideration can be an integral part of the database definition to various levels of extent. Behaviour is specified by user-defined operations on the database that are appropriate or relevant to the application domain. This explicit representation and storage of the definition of the database in a database schema is a distinguishing characteristic of a DBMS compared to conventional software where data definition is an integral part of the application program. A DDL compiler processes the schema descriptions into internal representations in the system catalogue.

The database can either be accessed by users directly, or indirectly, by other applications. In a direct user access of the database, the user usually states either ad hoc, or predefined queries (also termed "canned queries") to the database usually through a high-level query language such as SQL. Compared to a conventional and procedural-oriented programming language, a query language normally has a declarative nature. This means that you do not express the sequence or procedure for how to process your data, instead you declare what kind of data you are looking for. This is usually referred to as expressing "what" instead of "how". Ad hoc queries are transformed and optimized into an efficient and executable form by the query processor before they are executed, while predefined queries are compiled at definition time.

Indirect access of a database, through an application, can be made by including embedded DML statements or precompiled DML procedures. The DML includes constructs for retrieving, inserting, deleting, and modifying data in the database. Embedded DML statements are usually precompiled by a query processor before executing them whereas precompiled DML procedures are DML statements precompiled into a procedure, stored in the database, that can be called in the application.

Compilation and optimization of DDL and DML statements are made by the query processing tools that also interact with the system catalogue. An executable statement is thereafter delivered to the database manager that accesses the database to store or retrieve data. If the database is stored on disk this involves an interaction with the file manager to access the physical data. The database manager is also responsible for several other tasks in the DBMS, including the control of authorisation, concurrency, integrity constraint checking, and backup/recovery. authorisation controls the user accessibility of a database and can, for instance, restrict access privilege for a user to a specific part of the database. Concurrency control is responsible for controlling database interactions among concurrent users while preserving data consistency. Controlling data integrity involves keeping data consistent by checking that the specified consistency constraints are not violated. Backup and recovery are responsible for keeping data safe against failure usually by making periodical and persistent backups of the database and keeping a log of database operations. The ability to keep the database consistent is facilitated by defining database processing in terms of transactions. *Transactions* are operations on the database that represent atomic and controlled logical processing units.

3.1 CHARACTERISTICS AND OBJECTIVES OF DATABASE SYSTEMS

The main objectives of a DBMS include an efficient, flexible, reliable, and secure management of data. Certainly, the value and importance of these aspects vary among application areas as well as for specific purposes within application areas. For instance, in some administrative applications the efficiency of a DBS might be valued against manual data handling, while in computing-intensive engineering applications it must be compared to that of conventional file-processing applications. However, to meet these objectives a DBMS can include software tools that provide:

- Data modelling capabilities in terms of a basic data model. A data model includes a set of predefined constructs for structuring data that can involve predefined data types, basic operations, and user-defined data structures. The structure of data is further defined into a database schema including domain concepts, relationships, and even operations. A powerful and important aspect of data modelling is the ability of the user to define complex data objects and relationships. A system-supported data model also promotes and fosters the design and use of standardised and uniform data representations that will facilitate reuse, communication and exchange of data among applications and within organisations.
- A high-level database language, usually referred to as the query language, that provides the interface to the database. The DBL can be used directly and interactively for defining, manipulating and querying of data. DBL statements can also be embedded in a host language (the implementation language of the application) for indirectly accessing data in the database. In a DBL, data management is usually specified more declaratively than in a conventional programming language.
- Persistent storage of data, i.e. data (and program procedures) can be stored permanently on secondary storage and thus will survive termination of program execution and can later be retrieved. Transferring data between a DBMS and applications can give rise to what is called an *impedance mismatch* problem which has to be addressed. The impedance mismatch problem implies that the application and the DBMS have incompatible data representations, meaning that when data is exchanged it must be transformed. An approach to solve this problem is to be able to store and manipulate programming language objects (e.g. C++ or Smalltalk objects) persistently in the database which is the approach applied in some OO DBMS.
- Efficient accessibility of data. DBMSs support facilities for creating access structures, or indexes, that make access of data elements efficient. There are general indexing techniques, such as various tree data structures and hash tables, and techniques specialised for certain types of data, such as quad trees for spatial data. The DBMS also usually has facilities for optimizing queries, i.e. transforming a query into a form that has an effective execution order.
- Logical and physical data independence by views and language mappings. Data independence means that there is a separation between two software layers in such a way that there is no need for the “upper layer” to know anything about the data organisation or the data access techniques in the lower level. Data independence be-

tween the internal schema and the conceptual schema in a DBMS is referred to physical data independence since the conceptual schema needs no knowledge of physical data storage. Likewise, when the external schema does not need complete knowledge of the conceptual schema, it is referred to as logical data independence. It should be possible, for instance, to add information about an entity or add new entities to the conceptual schema without affecting the external schema. The DBMS accomplishes data independence by language mappings between different schemas and restricting certain operations to a specific schema. More complete discussions on issues in data independence are given in Date [83].

- Data sharing through concurrency control and transaction processing. Operations in a DBMS can be performed as transactions that are logical operation units on the database. The transaction processing software controls the state of transactions and guarantees that the database is always in, or can return to, a consistent state. Further, the DBMS can use concurrency control to ensure that several users can access and update the same data element in a controlled manner and guaranteeing a correct result. Thus, transactions and concurrency control ensure multiuser transactions to be performed correctly and several users can share data without bothering about interfering with other users.
- Access control through authorisation tools. Not every user might be allowed to access the complete database and to control this, the DBMS must include security and authorisation software. authorisation can, for instance, be specified in terms of privileged commands or software, or by allowing database access through a set of views.
- Different levels of fail security through facilities for logging, backup, and recovery. To assure that the database can be recovered from different types of software and hardware failures the DBMS include facilities for logging of transactions, making backups of the database, and recovery procedures to recover from failures and restore the database into its last consistent state.
- Another type of data security can be controlled to some extent by redundancy control, certain inconsistency control, and integrity constraints. Redundancy can be reduced by sharing data among applications. Limited redundancy can also be controlled by automatically propagating updates in the database to avoid inconsistencies. Even without redundancy there can be data inconsistencies. A data element can be updated with an inaccurate value in reference to itself or to some other data element. This type of inconsistency is controlled by adding *integrity constraints* in the database that can be controlled by the DBMS.
- Active behaviour can be introduced into the database through the addition of declarative rules and inference procedures (or deduction). By representing rules that should perform some defined action when some conditions are fulfilled an active behaviour can be achieved in the database.
- There is a well-known architecture for describing databases called the *three-schema architecture [84]*. Its aim is to isolate the physical database from applications. From the lowest level, the first schema is *the internal schema* that describes the physical

storage structure of the database in terms of the physical data model and access methods. The second level, the middle level, involves *the conceptual schema* and is a general conceptual representation of the problem domain. It describes entities, structures, and operators in terms of a data model without physical considerations. The third level represents *the external schemas* (or *views*) and describes subsets of the conceptual schema that are relevant to, and in a form suitable for, particular applications.

3.2 CONVENTIONAL DATABASE TECHNOLOGY

Various criteria can be applied for dividing DBMSs into certain categories. Usually, the underlying data model works as a base for such a division but you can also see division according to, for instance, usage profile, distribution, or structure of the DBMS. If we start with the data model, there are basically the hierarchical, the network, the relational, and the object-oriented data model.

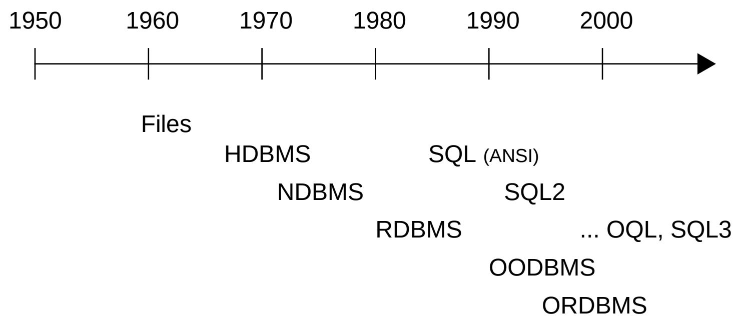


Figure 10. *The evolution of DBMS technology.*

3.2.1 Hierarchical database management systems

In hierarchical DBMSs data is arranged in hierarchical tree structures of connected record types. Data manipulation in the hierarchical model is accomplished by embedding data manipulation operators in a host language. There are a set of operators for navigating in the tree structure and operators for "updating" data in a record-at-a-time fashion. One of the earliest and most widespread hierarchical DBMS is IMS[1] (Information Management System).

1. IMS is a product of the IBM Corporation.

3.2.2 Network database management systems

Data represented in the network model is arranged in network structures represented by record types where interrelationships are represented by set types. A set type represents a one-to-many relationship between two record types. As for the hierarchical model, data manipulation is done by means of operators for navigating in the network and by record-at-a-time operators for updating data stored in the records. The data manipulation commands are also in this case intended to be embedded in a host language. The network model is often referred to as the CODASYL[1] (Conference on Data Systems Languages) model and an example of a commercial CODASYL-based DBMS is IDMS[2].

3.2.3 Relational database management systems

The relational data model is based on a single uniform data structure called a *relation* that is used to represent both data and interrelations. A relation is commonly viewed as a named *table* where rows are called *tuples* and where columns contain *attributes*. A table can represent both a class or a relationship between classes. When the table represents a class, the tuples represent specific instances, of that class and the attributes represent common properties for the class. Similarly, a table can be used to represent a relationship, whose instances are represented by rows in the table. In this case, the columns represent the participating entity types. Further, relationships can be of n-ary type where n states the number of columns in the relation.

Different types of constraints can usually be defined on relations in order to control the consistency of data. The different types of constraints include domain constraints, key constraints, and entity integrity and referential integrity constraints. These are used to control that the value *domain* of attributes is correct, that tuples are uniquely defined, and to maintain consistency among tuples.

Data manipulation on the relational model is usually made through a high-level query language, usually SQL, that includes both procedural and nonprocedural constructs. The term query language associates only to retrieval operations, but in addition, a general query language usually includes operations for data definition, and update. For updates of relations, there are three basic operations for inserting, deleting, and modifying tuples. Section 3.8 presents more about data management in R DBMSs and SQL.

Compared to older data models, such as the hierarchical and the network models, the relational model is more independent from its physical implementation. It also has a more formal basis since its simple and uniform base model has made it possible to de-

1. Originally defined by the CODASYL Data Base Task Group in 1971.
2. IDMS was originally a product of Cullinet Software, Inc., and is currently marketed under the name CA-IDMS by Computer Associates.

rive a relational algebra and a relational calculus that is based on mathematical theories. An important reason to the success of R DBMSs, is the availability of declarative query facilities in SQL.

Examples of well-known commercial R DBMSs are ORACLE[1], Sybase[2], INGRES[3], INFORMIX[4], and DB2[5].

3.3 OBJECT DATABASE TECHNOLOGY

Classical DBMS technology concentrated on supporting administrative applications. However, in contrast to these traditional administrative database applications, scientific and engineering applications usually involve models and analysis methods of higher complexity, Cattell [85]. This has put new requirements on existing database technology including support for domain-specific data modelling capabilities while preserving efficient data processing. It is expected that the next generation of object database technology, where OO and relational database technologies are merged, can meet several of these needs. This also includes the availability of extensibility and main-memory residency in DBMSs that are here presented as separate subsections.

3.3.1 Object-oriented concepts

Basic OO concepts that usually are supported in OO DBMS include objects, object identity, composite objects, methods, encapsulation, type hierarchies and inheritance, operator overloading, late binding, and version and configuration management.

- *Objects* form a fundamental concept of OO methodology. An object represents a physical or abstract entity that possesses certain characteristics. Similar objects that share the same characteristics can usually be defined and referenced as a group which is a means to introduce structure and reduce complexity in a concept domain. Other fundamental concepts of OO methodology including encapsulation, classes (or types), and inheritance are described below. In contrast to objects implemented by means of OO programming languages, OO databases provide facilities for handling persistent objects, i.e. objects that persist after finishing the execution of a program. Database objects can also be shared by several different applications.
- *Object identity* (OID) is a unique system-generated identifier for an object. By letting the system handle object naming and look-up, these facilities can be made more efficient and relieve the application programmer from implementing this mecha-

1. ORACLE is a product of Oracle, Inc.
2. Sybase is a product of Sybase, Inc.
3. INGRES is a product of Ingres, Inc.
4. INFORMIX is a product of Informix, Inc.
5. DB2 is a product of the IBM Corporation.

nism. OID:s are used to refer to specific objects in the database (e.g. for making associations between objects) or in the application program.

- *Composite objects* are objects that are built up of other objects, possibly in several layers. For instance, a mechanical design can include many design details. Grouping of objects is a kind of aggregation that can be represented with ordinary relationships in the database. However, some DBMSs provide special treatment of aggregation relationships including such facilities as deleting, copying, and clustering of composite objects. Composite objects are also referred to as *structured complex objects* in contrast to the term *unstructured complex objects* that are used for referring to large binary objects (BLOBs) such as images, sounds, or unstructured texts.

- *Attributes* represent named properties of objects and can have various representations. *Simple attributes* are used to describe properties that can be represented by literal values such as integers, reals, or strings. *Complex attributes* consist of references, collections, and derived attributes. *Reference attributes* represent relationships between objects through OID:s. *Collection attributes* represent groups of simple attributes or references in the form of arrays, lists, sets, and multi-sets. *Derived attributes* are represented in terms of other attributes and the retrieval of attribute values involves a computation. Some systems, like AMOS, have capabilities to automatically represent *invertible attributes*, i.e. attributes can be used in both directions. Normally, this capability must be represented by two relations in each direction that must be kept consistent. Attributes are used for representing different types of relationships in OO DBMSs. Object attributes offer the most direct technique for representing properties in pure OO methodology. In fact, when considering real world properties in more detail, one realises that properties are actually relations between objects. For instance, the hardness of wooden table is related to the exposed object such as a hand or an axe. For the hand the table is hard, but to the axe it might be soft. Another example is colour that at first thought is closely related to a thing but that is really dependent on the watcher's ability to see colours. In practice, we usually relate these properties to one specific type of object but sometimes a more general modelling is more convenient. By introducing functions overloaded on all arguments, Flodin et al. [19], this type of more general modelling technique can be used to describe relations dependent on several object types.

- Storing *database methods* associated with database objects in the database is an important ability of OO DBMSs. However, first generation OO DBMSs do not normally support the ability to store methods in the database; instead they are stored outside the database. In many situations, domain data is more conveniently represented by procedural information. This can, for instance, be a weight function that is derived from the current geometry of a design or function that calculates the current position or velocity of a design component. By allowing procedures in the database they can be used in filtering out and reducing the amount of data that should be passed to the application. Procedural information has normally been encoded in the application programs but the evolution of OO database techniques has facilitated storage of procedures in the database. In pure OO systems all communication with objects is performed through methods that hide the internal structure of the objects.

This is called *encapsulation* and provides one form of data independence.

- The structure and behaviour of objects (called the intent) are defined by *types* (or *classes*) that are also used for controlling the type extent, i.e. the set of objects that adhere to a certain type. Objects are used for representing types in a meta-level manner and define the type intent or object properties as attributes and methods. OO methodologies usually involve some *generalisation* mechanism for types, providing capabilities to structure types into hierarchies or acyclic graphs. These structures define sub- and supertype relationships among types and can be used for applying various hierarchical relationships. For instance, it can be used for *inheritance* where a subtype inherits properties from its supertypes. When a type inherits from several parallel supertypes it is called *multiple inheritance*.
- *Operator overloading*, sometimes called ad hoc polymorphism, is one kind of polymorphism that plays an important role in OO modelling. *Polymorphism* implies an ability to have several forms. In the case of computer science, this term is used to denote various forms of sharing the same name or reference for different programming objects like variables and functions. Operator overloading is one form of polymorphism that permits operators to have different implementations for different types, or more generally for the signature for the operator. This can be accomplished by different combinations of static or dynamic type checking, and early binding (also called static binding) or late binding (also called dynamic binding) of operators. The term *overriding* is sometimes used to refer to the situation where a type has an inherited operator name and redefines its implementation. This is a distinction from the more general overloading term where inheritance is not required. Examples of advanced use of overloaded functions for modelling linear algebraic matrix operations are described in [19]. Efficient treatment of overloaded functions in AMOS is presented in Flodin and Risch [86] and Flodin [87].
- System-supported *versions* and *configurations* facilitate the representation and management of multiple variants of the same object in the database. A version is a variant of a single specific object whereas a configuration is a consistent variant of a complex object. A configuration of a complex object includes other objects that in turn have their own (possibly different) versions or configurations. Versions and configurations can be used to keep different copies of, for instance, designs, to control authorisation to different versions of objects, and they can also be used as tools for concurrency control, i.e. to coordinate work between multiple users. Several OO DBMSs have basic features that support version and configuration management.

3.3.2 Object-oriented and object-relational database technology

Object database technology has evolved from two different areas. Both these strategies to provide object-oriented functionality in DBMSs share some characteristics but also have major differences. What is referred to as *the first generation* OO DBMSs, Atkinson et al. [88], evolved in the area of OO programming languages where the need for database facilities in OO applications arose, such as the ability of storing programming objects (e.g. C++ or Smalltalk objects) persistently on secondary storage. When this kind of application terminates, objects persist on secondary storage and can later be retrieved by another application. First generation OO DBMSs also usually include basic database facilities such as a simple query language, access techniques such as hashing and clustering, transaction management, and concurrency control and recovery. However, they are incompatible with R DBMSs and do not include several R DBMS features such as a complete declarative query language, meta data management, views, and authorisation. Their advantage is a seamless integration with their corresponding OO programming language. Products originating from the first generation OO DBMS approach are Gemstone[1], O_2[2], Objectivity[3], ObjectStore[4], ONTOS[5], and Versant[6].

Systems called *the second generation* OO DBMSs evolved from the classical relational database community and were also inspired by OO ideas. The attempt to meet the needs required by new types of database applications, as for instance from the scientific and engineering area, has resulted in an extension of relational database technology with OO capabilities. Examples of these capabilities include object identity, object structure, composite objects, type constructors, encapsulation, inheritance, and OO extensions of a query language. By combining this type of system with a call-level interface, it is possible to provide OO DBMS capabilities to programming language-based applications. These DBMSs were first referred to as extended-relational DBMSs, Stonebraker et al. [89], but recently the term *object-relational database management systems* (OR DBMSs), DBMS [4], Frank [5], and Stonebraker and Moore [6], has gained considerable acceptance. Examples of this type of product are Odapter[7], Illustra[8], and UniSQL[9]. The research prototype AMOS, that is used in this work, is based on this approach. An important aspect of OR database technology is the availability of several kinds of extensibility including query language, query processing, and storage management extensibility. These issues are discussed in the next section that covers extensible database technology.

These two approaches to accomplishing object database technology are currently blended, Kim [90], in that the first generation OO DBMSs are extended with, for example, query languages, query optimization, views, constraints, meta data management, authorisation, triggers, transaction management, two-phase commit, and parameterised performance tuning. Likewise, the second generation OO DBMSs are extended to incorporate OO concepts to be able to handle objects and OO modelling. A detailed analysis of the requirements on query processing in object-relational database technology is

1. Gemstone is a product of GemStone Systems, Inc.
2. O_2 is a product of O_2 Technology.
3. Objectivity is a product of Objectivity, Inc.
4. ObjectStore is a product of Object International, Inc.
5. ONTOS is a product of ONTOS, Inc.
6. Versant is a product of Versant Technology, Inc.
7. Odapter is a product of Hewlett-Packard, Inc.
8. Illustra is a product of Illustra, Inc.
9. UniSQL is a product of UniSQL, Inc.

provided in [6].

3.4 EXTENSIBLE DATABASE TECHNOLOGY

Extensible database technology has been identified, by Carey [91] and Carey and Haas [92] among others, as a key technology for providing database technology to new emerging database applications, such as advanced engineering applications. Applications of this type usually require more powerful modelling techniques and performance than provided by standard DBMSs. This calls for database techniques that can be tailored to support new data types, functions, complex objects, storage and access techniques. According to Carey and Haas, another motivation for this approach is to facilitate the incorporation of emerging database technologies into existing DBMSs. This motivation can actually be generalised into a motivation to provide emerging database technology (or software technology in general) to existing applications. Most engineering applications are implemented in a conventional programming language which means that large parts of the application must be redesigned and reimplemented in order to take advantage of new software technologies. It would be much easier to take advantage of new software technologies if, instead, a software tool was used to develop the application. An extensible DBMS can, for instance, provide mechanisms for concurrency control and for distribution and replication of data, that would immediately be available to the application developer.

Carey and Haas [92], have identified three levels of extensibility that are applicable in extensible DBMSs:

1. *Data storage and access methods extensions*. It must be possible in extensible database technology to introduce new storage structures and indexing techniques for data. Applications are very different in their needs for specialised data structures and access methods. For instance, numerical data is used very differently among (or within) applications. Some concepts are represented and handled as single numerical values that maybe should be retrieved, compared with similar data, or updated with new values. This could be a hole diameter or a single temperature. In contrast, other (and even the same) concepts are more efficiently represented as large collections of numerical data, usually in the form of arrays or matrices. Perhaps the collection should be a discrete representation of a concept that is temporally or spatially distributed, such as a temperature field or a stiffness distribution. These concepts are usually involved in more complex mathematical operations that can be efficiently implemented using array and matrix data representations.

2. *Query language extensions* that include the ability of defining *abstract data types* (ADTs), i.e. the definition of new data types and their corresponding operations. In OO technology this is accomplished by allowing the user to define new types (or classes) of objects and defining operations on these types. This could even include

the ability to define type constructors, i.e. an ability to have parameterised type definitions, discussed in Werner [93]. Extensibility at the query language level also includes more general possibilities to extend the query language, such as defining operations for data collections. This covers different kinds of aggregation operators such as summation of numbers, counting members, or calculating averages.

3. *Query processing extensions*. The query processor can also be extended with new execution strategies and new methods for combining existing operators. One might wish to include better algorithms for join operators or new strategies for operator ordering. Furthermore, the query processor must be able to handle the new application-specific operations that the query language has been extended with. Hence, it must be possible to add cost measures and optimization rules for these new operations that should be treated by the query optimizer.

Extensibility is handled to various degrees by different DBMSs categories. In pure OO DBMSs, the extensibility is accomplished only through user-defined types and methods. This type of extensibility can also be available in DBMSs based on extensions of relational database technology but these can also provide extensible storage managers and query processors. There exist several research systems in the extensible DBMS field built upon different basic data models. In Starburst, McPherson and Pirahesh [94], and in Postgres, Stonebraker et al. [95] (commercialised as Illustra, Stonebraker and Moore [6]), the data models are built on extensions to the relational data model. Further, the PROBE system, Goldhirsh and Orenstein [96], and the ORION system, Woelk and Kim [97], are built on an OO data model. In AMOS, Fahl et al. [8], the OO data model has been combined with relational query capabilities. GENESIS, Batory [98], and EXODUS, Carey and DeWitt [99], have a somewhat different approach in providing toolkits for generating application-specific DBMSs.

The extension of a DBMS with new data structures, operators and suchlike must also handle the interaction with other database facilities, such as security, concurrency control, and recovery. Another problem is the interface that should bridge the gap between the extensible DBMSs and the application that might be implemented in a language with a more primitive type system. Some efforts have been made in this area within the ODMG standardisation where the OQL[1] language, Cattell [17], is defined to permit different language bindings for representing the data types of OQL. Using the extensible DBMS as a powerful tool for actually implementing applications adds another view to the interface problem. In this case, the application functionality is partly, or to a large extent, implemented within the DBMS where more general resources can be “plugged into” the DBMS as modules. This raises the questions of which data structures should be provided by the DBMS and where the processing should take place. By delegating certain application-specific operations to the DBMS one could take advantage of efficient data access techniques of the DBMS. This approach could further reduce the over-

1. In OQL, user-defined methods can be included in queries but they can not be handled by the optimizer.

head for transporting and transforming data between the DBMS and the application.

3.5 MAIN-MEMORY DATABASE TECHNOLOGY

Main-memory (MM) DBMSs have evolved during the last few years, DeWitt et al. [100] and Eich [101]. Conventionally, databases are *disk-based*, i.e. they reside on a secondary storage media, usually in the form of magnetic harddisks. However, hardware developments such as large main-memories and fast networks have provided potential for new design ideas. This has cleared a path for the development of MM DBMSs that assume that the entire database resides in primary memory.

There are well-known differences in the characteristics of secondary and primary memory, Garcia-Molina and Salem [102], that include:

- Main-memory has orders of magnitudes lower access times than magnetic disk memory.
- Main-memory is usually volatile while magnetic disks are not. However, non-volatile main-memories are becoming more common.
- Magnetic disks have a high initiation cost for each access that is independent of the amount of data that should be retrieved. This fact makes this kind of storage devices block-oriented. Main-memory has low initiation cost and consequently needs no block-orientation.
- Sequential access is faster than random access for magnetic disks. This does not hold for main-memories since they are really random access memories. Thus, data organisation and clustering are more important for disk storage.
- Main-memory data can be more vulnerable to software errors than disk-resident data since main-memories can be directly accessed by processors.

Disk-based database systems cache data from secondary to primary memory for processing. In contrast, an MM DBMS can have a copy of data on disk for backup purposes. Hence, both architectures have data located both in main-memory and on disk but MM DBMSs have their principal copy in memory while disk-based DBMSs have their principal copy on disk. These differences considerably influence the design for these two database system architectures. Since disk access is the bottleneck for disk-based systems, they are designed with data structures, access methods, and clustering techniques that should minimise disk access for reading and writing data to disk. Main-memory database systems access data directly in memory which results in better performance than conventional disk-based database systems. For main-memory systems it is instead the processing time that is the origin for the performance. Garcia-Molina and Salem [102] mention performance considerations as especially important to real-time applications. However, performance can also be of vital importance in other engineering tasks such as interactive engineering applications for design and analysis that involve computing-intensive calculations and simulations. Performance is not only accomplished through main-memory residency; at least as important is query optimization that can reduce the combinatorial complexity of queries by several orders of magnitudes. Litwin and Risch [103] show that a combination of main-memory residency and query optimization is required for best performance. This technique can actually reduce execution time to less than one disk access in some situations which would be impossible with traditional disk-based techniques.

Access methods and data representations are additional issues that are effected by switching from disk-based to main-memory-based technology. Specialised access methods are utilised to take advantage of main-memory characteristics. For instance, there are methods based on hashing techniques, special tree representations, or pointer-based indexing. Data representation techniques must also be adapted for main-memory. Furthermore, the query execution cost is determined by the processing cost instead of the cost for disk access as in disk-based systems, as discussed in more detail in Listgarten and Neimat [104]. Concurrency control, transaction and recovery processing, and data clustering and migration are also mechanisms that need specialised treatment, Garcia-Molina and Salem [102]. The techniques for exchanging data between applications and the database also need adaption to main-memory residency of the application programming interface.

MM DBMSs, are exemplified by SmallBase, Listgarten and Neimat [104] and Heytens et al. [105], WS-IRIS [103] and AMOS [8]. SmallBase is built upon a relational data model, whereas WS-IRIS and AMOS are based on an object-relational data model. A main design idea behind main-memory, as well as extensible, database technology is the ability to use these types of DBMSs embedded within applications.

3.6 ADDITIONAL DATABASE TECHNOLOGIES

3.6.1 Distributed database management systems

DBMSs can be centralised or distributed. A centralised DBMS resides on a single computer site that includes the complete database and the DBMS software and hardware. Distributed database management systems [106], D DBMSs, take a contrasting approach, i.e. the data and software are distributed over a hardware platform that consists of multiple computer sites that are connected by some form of communication network. There are different potential benefits that motivate the use of D DBMSs that include:

- Distributed databases can naturally conform to distributed applications. Database applications may be physically distributed (different locations) or logically distributed (different application domains) that require or make it convenient to distribute data over several sites.
- The reliability and availability can be increased since redundancy can be introduced

by replicating data and functionality over several sites.

- Controlled data sharing can be provided by keeping local control over data but allowing distributed access.
- Performance can be increased due to the distribution of transactions over multiple sites that would otherwise be processed at a centralised site.

Distributed database systems can further be divided into various categories depending on additional factors such as the level of homogeneity, autonomy, and transparency.

Distributed database systems can have different levels of homogeneity. In a *homogenous database system* the same DBMS is used on both servers and clients, which is not the case for *heterogeneous database systems*.

There can further be different levels of autonomy in D DBMSs. At one end of the spectrum we have no autonomy where a D DBMS acts as a centralised DBMS with a single conceptual schema and the access must be made through one server. At the other end we have *federated DBMSs* or *multidatabase systems* where each server has an independent DBMS. It is then possible to have local databases and transactions that provide a higher degree of autonomy. Global clients can then, through a multidatabase interface, access privileged parts of the collection of autonomous databases (that form the multidatabase).

Another aspect of distribution of databases is the degree of *distribution transparency* of the system. Does the user need to now anything of where data is distributed and of how data is fragmented or replicated? In a transparent system this knowledge is not needed but is required to some extent in systems with a lower transparency level.

Technology for D DBMSs can be of great importance in applications for engineering design and analysis where different engineering domains withhold their own databases. Certain parts of that data must be shared among several disciplines and further be kept in consistence at the global level. The role of D DBMSs in engineering applications is briefly discussed in sections Section 3.7, Section 6.2, and Section 5.1.

3.6.2 Active database management systems

Database systems have traditionally been passive, i.e. the communication between the users or applications and the DBMS have been triggered by the former. In contrast, an *active database management system* can in itself react and act on database events (e.g. database updates) or external events (e.g. updates of physical signals). Active functionality is commonly accomplished by providing some form of rule system in the DBMS. *Rules* usually consist of a condition part and an action part. The condition part defines the event situation that should be true for executing the operations defined by the action part. Actions can consists of database operations and external operations. When the rule system detects relevant events it triggers the condition monitoring and, if the condition part evaluates to true, the action part is executed.

The active database functionality can be used to provide consistency control as an integral part of data management. For instance, constraint management can use rules that detect and abort transactions that attempt to perform updates that violate database constraints, Ceri and Widom [107], Kim [90], Risch and Sköld [108], Sköld [109], Sköld and Risch [110].

Explicit declaration of rules for data consistency and consistency handling within the DBMS can increase the flexibility of consistency control and the reliability of data. By integrating the rule system within the DBMS, the data exchange between an application and a database can be reduced and, furthermore, query optimization techniques can be used for accomplishing efficient rule execution.

The ideas of active DBMS technology are influenced by the fields of knowledge-based and expert systems, expert database systems, and deductive DBMSs. Techniques from these fields, like rule-based reasoning and inference techniques, have been adapted to suit needs emphasised in the database area, such as handling large data sets and efficiency-related considerations.

Active behaviour in AMOS is treated in Risch and Sköld [108], Sköld [109] and Sköld and Risch [110], that describe the extensions of AMOS with efficient change monitoring techniques using active rules. These techniques could, for example, be used for constraint checking in model preparation and result evaluation and for controlling solution parameters in the engineering analysis process.

3.7 SCIENTIFIC AND ENGINEERING DATABASE TECHNOLOGY

The database field can also be divided into different application areas where the requirements on the database technology can vary heavily. The field of scientific and engineering database technology is the most relevant for this work.

Classical DBMS technology was primarily designed for conventional applications of business and administration that were characterized by large amounts of data with relatively simple structure, and where the operations on data were small and simple as well. However, as for instance pointed out by Cattell [85], these techniques have had less to offer to more complex software applications for supporting mechanical engineering, electrical engineering, document handling, software and systems engineering, process control, science and medicine, expert systems, and advanced financial applications.

The field of scientific and engineering applications has been especially emphasised by several authors, Cattell [85], Navathe and Elmasri [80], Korth and Silberschatz [111], Loomis [81], and Ullman [112], as an important field for future database support. In the literature there exist a division of database applications into scientific databases and engineering databases. The term “scientific” is usually associated to areas such as life, earth, space sciences and suchlike where large amounts of “real” data (experimental or measured data) is handled. On the other hand, “engineering” refers to typical engineering issues of producing articles from mechanical or electrical components (such as nuts, bolts, and resistors) to complete complex products or systems (such as cars, aircraft, power plants, and computers). The engineering process can in itself be quite complex including several different subprocesses that should be coordinated. For instance a mechanical design should be specified, designed, analysed, manufactured, tested, maintained, and possibly re-cycled; manufacturing should be prepared and manufacturing tools should be produced, for instance. A simple product might involve just a few people within an enterprise while a more complex product requires the engagement of many employees with different skills that might span several enterprises. In the engineering field a large amount of data is “synthetic” (that is generated by design tools as CAD software, or simulation tools).

More specifically, conventional database applications were, according to Korth and Silberschatz [111], characterised by:

- Uniform and equal-sized data items.
- Record-oriented and fixed-length data items.
- Small data items.
- Atomic fields. Record-fields are unstructured and of fixed length.
- Short transactions with an execution time of fractions of a second.
- Conceptual schema changes are infrequent and usually involve only simple modifications.

Much of the effort in research and development of new database technology aims at supporting those kinds of new emerging database applications mentioned above. Compared to classical database applications, these applications usually deal with a more complicated problem domain. They require the representation of a much broader spectrum of domain information that involves complex composite concepts of several types and of non-uniform data items, complex concept structures, associated operations, and domain knowledge. The conceptual structure also evolves during its life-cycle meaning that the schema will be exposed to more changes. Further, the usage profile can also be quite different as data should usually be shared among several project members in a computing environment. Specific data processing activities typically have much longer duration in these applications and data exists in several versions which must be kept consistent. The application part will usually contain much functionality and it is of major importance to have a good programming language interface to achieve efficient communication between the application part and the DBMS. It is also required that the database application has a performance of, at least, the same level as a conventionally implemented application. Specifically in engineering applications, data usually incorporates the following characteristics:

- Engineering designs are in general represented by collections of non-homogeneous data objects. A specific design object is typically a composition of other design objects that form a more complex structure than those found in conventional DBMS applications. Further, these object compositions usually include many domain-specific relationships. These relationships can be of various types, such as mechanical, mathematical, spatial, and temporal relationships, and are important in engineering applications for representing the structure and functionality of the design. In addition, this type of relationships might need special treatment in the database concerning data structures and access techniques.
- Derived data, i.e. data computed from basic data elements, is important in engineering applications. This type of data can require a recalculation whenever any underlying data element is updated. For instance, the weight of a design and any other quantity dependent on the weight must be updated whenever the geometry is changed.
- Engineering data involves many domain-specific types where each type does not need to include a large amount of instances. However, a complete design can in its entirety represent a substantial amount of data.
- Engineering domains include a lot of domain knowledge that governs the permissibility and consistency of data. At present, this type of knowledge is usually embedded within applications or explicitly handled be users. By extracting domain knowledge, from applications and users, and representing it as meta-knowledge in the database it will be represented more explicitly and become easier to manage.
- Engineering data objects are usually built up by several basic data types for literal, numerical, and image data (BLOB's). Numerical data is a central component of engineering data and is usually used in aggregate data items such as in arrays, vectors and matrices. These numerical data types are used in describing higher-level engineering elements, such as geometries formed by complex compositions of basic geometry elements. Conventional commercial DBMSs do not normally support data structures suitable for engineering applications; they mainly support basic data types such as character strings, integers, and floats (not always). However, these systems will evolve and provide more complex data types as well and with the introduction of OO DBMSs, this fact does not hold since these usually support aggregate and user-defined data types. On the other hand, this will probably not be enough. Advanced engineering applications will always demand specialised data types, such as compact and efficient matrix representations optimized for specific applications. Extensible DBMSs are able to provide this functionality.
- The domain-specific functionality and behaviour are important aspects of engineering design that one would like to capture in the database. For example, how a certain mechanical connection is allowed to move and if it is dependent on other parts in the design. Representing functionality and behaviour usually includes some kinds of computations that can be represented by allowing the representation of functions

and procedures in the database.

- Engineering data structures are in general more exposed to change compared to data structures in conventional DBMS applications. Engineering designs evolve continuously during the development process in comparison to conventional database applications where the structure of data is more static.
- Transactions in engineering applications are typically of longer duration than normal database transactions that last for fractions of seconds. Design and analysis activities may take hours, days, weeks, and even longer to perform. Commercial DBMSs use locking-techniques that assume transaction times of a few seconds. However, specific engineering activities can include smaller transactions like updates of a local design that must be kept consistent. Thus, engineering applications can involve the complete scale from short to long transactions. The profile of updates of engineering design is also quite different from conventional updates that typically should update single values. It is not uncommon that updates in engineering applications are far-reaching since many geometrical, topological, and functional relationships involve many related objects. Changing a single design object can imply updates of many related design objects.
- Engineering design and analysis systems must provide configuration and versioning support. A design passes through several design stages where it can exist in several related versions and where each version has its own state, specifying if it is released or is in any other relevant state. The design support system must also permit engineers to work on their own private or local design that is extracted out of and can be merged into a public or global design representation.
- Engineering computing environments can be of various complexity from single personal computers to large networks of work stations and including special servers for data storage (DBMS servers) and processing (super and parallel computers). At least in the complex case, data can be distributed over the complete network where responsibilities for specific parts are distributed on specific organisational functions. For instance, there might be different persons who are responsible for geometry data, materials data, and test data. The data that represent the current and complete design can therefore be distributed over the network and subsections of data do not need to be in the same state at any time. There can also, as already mentioned, be local copies of specific design objects or other associated objects such as various analysis models. For instance, for performing some type of mechanical analysis you might need to retrieve geometry, material, and loading data from different sources into your analysis model. The analysis can thereafter be carried out and perhaps iterated before a satisfactory result has been accomplished. The iteration process might require alterations of data and data exchange between the local and global level. At certain “check-points”, inconsistencies between these levels must be resolved and updated. Thus, DBMSs might be involved for data management on both the local and global level for sharing, exchanging, and combining data and for maintaining consistency between levels.
- Engineering designs can include a lot of repetitive data in forms of nuts and bolts,

form features like holes, fillets, or notches. For this type of design object, only distinct values need to be stored for every instance and general properties can be reused.

- Engineering applications usually have higher requirements on execution performance than conventional applications. Data is typically used in highly interactive and computing-intensive tasks with high performance requirements that should be matched by the DBMS as well.
- The amount of engineering data in a design project can be huge. These levels are expected to rise since more design, analysis, testing, and manufacturing activities are being computerised and more data is being generated in each activity. DBMSs for engineering applications must be able to cope with these amounts of data.

Computational mechanics forms a sub-area of mechanical engineering where efficient computing support is obligatory. In this area, finite element analysis is utilised for analysing and simulating different phenomena such as mechanical, thermal, fluid, or acoustic. FEA software should handle large complex analysis models and algorithms in a computing environment that involves several other software tools. In this area, database technology can be introduced for a number of reasons and tasks including:

- The administration and management of FEA models and simulation results, both locally or globally. This involve administrating geometry, material, and load data, complete analysis models, analysis results, and the relations between FEA data and corresponding design objects. There are also other administrative data not directly related to FEA, such as design project data, that can be relevant for this task as well.
- Sharing, exchanging, combining, and transforming data among different tools and users. The same basic data, such as data regarding the geometrical form of the design, might be used by several users or tools. If different tools use the same data but in different formats, data must be transformed to the appropriate format for specific tools. The DBMS can provide a uniform and standardised data representation that suits different tools. Furthermore, a DBMS can be a powerful medium for exchanging data between tools. Geometrical data generated in a CAD program might be combined with load data from some load handling program, and it can be provided to an FEA program by means of a data-based representation and retrieval.
- Extending the domain functionality of FEA applications. Additional functionality can be added to FEA systems by representing and utilising domain knowledge, such as different types of constraints on, or rules for, the FEA domain. For example, the DBMS can check if finite elements are too distorted or if calculated stresses exceed some critical level. Some commercial DBMSs currently provide mechanisms for defining rules that can detect and react when a certain condition is fulfilled in the database.
- Extending the system functionality of FEA applications. If an FEA application is tightly integrated with or developed by means of a DBMS, the final application can take advantage of generic DBMS techniques for data management including data

sharing, distribution and replication of data, concurrency control, transaction processing, persistency, backup, and recovery. Furthermore, the DBMS provide the application with a data model, storage management, and a query language and processor that support application modelling.

- Development, evolution, and maintenance of FEA software. By using generic software developments tools, such as a DBMS, applications can be defined at a higher conceptual level than is normally done in conventional applications. Data is modelled and manipulated at a higher level resulting in more flexible, composable, and reusable models. Generic subsystems of the DBMS (see previous item) can be used directly and need not be re-implemented and a great deal of low level implementation details can be avoided. This will restrict, isolate, and reduce complexity of those parts of the application that concern the actual problem domain resulting in an application that is easier to maintain, evolve, and further develop.

The items concerning FEA and listed above can to various degrees be accomplished with database technology of today. In general, the first three items can involve a DBMS that is loosely coupled to the FEA program. Data can be stored in and retrieved from the database but most of the application functionality is kept within the FEA program. The DBMS will complement the FEA program by supporting pure DBMS facilities. The latter two items involve a more tight coupling between the FEA program and the DBMS. In this situation, database technology is used for developing, and to be an integral part of, the FEA application. Domain-specific functionality is here transferred from the FEA program to the DBMS. Various architectural considerations for tightly coupled systems are further discussed in Section 5.1.

Database technology for scientific and engineering databases has, in for example [1] [2] [3], been identified as an area where further research is needed to develop database technologies that meet the requirements of these applications. Much attention has recently been paid to OO DBMSs which are considered to be specifically suitable for this type of application. A number of characteristics that favour object-oriented models have been pointed out by Navathe and Elmasri [80] and Cattell [85]. Object-orientation provides modelling that is isomorphic to the problem domain, provides complex model structures, has uniformity of data access, and provides extensible models resulting in a higher level of modularity and flexibility of the models. The referenced and additional mechanisms that OO DBMSs provide for supporting advanced data management include:

- Unique system-provided object identifiers are convenient in engineering applications that typically deal with unnamed design objects such as point, lines, and surfaces in a geometrical model.
- Composite objects are everyday things in the engineering world.
- Referential integrity is important for maintaining consistency within complex engineering models.
- Object-type hierarchies provide an important mechanism for structuring engineer-

ing data.

- Ability to store associated procedures in the database supports representation of domain functionality.
- Object encapsulation can hide implementation details and provide limited data independence, but does also restrict the expressibility that can be inconvenient in certain situations.
- Ordered sets and references are very useful in engineering applications where the order among objects is used in many situations.
- Large data blocks are applicable, for instance for representing large numerical matrices.
- A programming language interface is important for integration with applications and for representing domain operators.
- Multiple database versions must be supplied for representing engineering design versions.
- Long-term lock and check-out mechanisms must be supplied to conform to engineering design tasks.
- Efficient remote and local database access is important since engineering models include large amounts of data that should be processed and presented to the user.
- Single-user performance is very important since most engineering applications presuppose interactive computer-intensive usage.
- Ease of schema changes supports the representation of evolving designs.

However, DBMSs for new advanced applications also need the support of traditional database facilities, which is emphasized by Cattell [85] and Stonebraker et al. [89]. The requirements that engineering applications would benefit from are here presented with comments on the relation to the previous discussion on engineering data, and are as follows:

- Ability to store and handle large volumes of structured data persistently.
- Access to query language facilities. A query language in its general meaning, including data definition and manipulation is a powerful tool to model and manipulate domain data. Advanced applications, such as in the field of computational mechanics, require extensible query languages that permit the definition of domain operators.
- Tools for designing application user-interfaces. Traditional forms-based interfaces might be useful for administrating engineering data but are not for design and analysis applications where much data is distributed over space and time. However, new advanced GUI tools might support the design of form- and geometry-related interfaces that can be useful.

- Capability of distributed databases can be important for storing engineering data on several servers in a network.
- Support for triggers or more advanced active rules can be useful for specific tasks such as automatic checking of analysis results and other design constraints.
- Fine-grained concurrency can be required along with long-running transactions in engineering applications.
- Crash recovery capability for keeping an engineering database consistent is desirable.
- Data independence is an important aspect since it is a basic mechanism for sharing data among applications and it is a means to raise the conceptual level in data modelling.

When the application of database technology in advanced engineering applications spreads, the requirements on DBMSs will be further widened. One such requirement is the ability to do analysis and synthesis of data in the database, i.e. data should be processed, analysed, and synthesised by domain-specific database operators instead of transferring data into the application for processing. This has already received attention in other areas such as in geographical informations systems (GIS) [113], chemistry [114], and advanced financial applications [115].

For applications that require specialised data structures, perhaps in combination with advanced processing algorithms, extensible DBMSs can provide key technology for implementing these applications. The DBMS can be extended with new data structures and access techniques that can be seamlessly managed by the system. One example would be numerical analysis algorithms that are usually based on compact and efficient matrix representations, such as skyline or sparse matrix representations. These specialised representations need dedicated operators for numerical algorithms in linear and non-linear algebra, and mathematical programming, for example. This approach will be further discussed in Section 4.1.

A closing comment regarding scientific and engineering data management; there is really no clear distinction between the areas of science and engineering. The engineering field also involves much “real” data, such as test data from different activities, and scientific work involves simulated data as well. Further, many computing tools are used in both science and engineering. Programs for FEA are used in industrial product development as well as in research in the computational mechanics field and others.

3.8 QUERY LANGUAGES FOR DBMS

The term *query language* is commonly used as a synonym for *database language (DBL)*, i.e. the language interface to the DBMS. These terms where introduced earlier in this chapter but will here be presented in more detail and complemented with some general issues and directions of query languages. Query languages are usually higher-level languages than ordinary programming languages. The query language is the actual data management language that is used for defining, updating, and querying data in the database. Sometimes, the term query language is only used to refer to the pure query language, i.e. the part of a database language that concerns querying of data and not data definitions and updates. It is also quite common to view the database language interface as a collection of languages for different purposes that include a *data definition language (DDL)* for data definition and schema management, and a *data manipulation language (DML)* for updating and querying the database. The data definition is sometimes further separated into a DDL, a *storage definition language (SDL)* for defining the physical storage schema, and a *view definition language (VDL)* for defining database views.

General query languages were first developed for R DBMSs and included SQL, the most well-known query language that has become the standard query language for R DBMSs. Other examples of relational query languages are QBE and QUEL.

The field of OO DBMSs still lacks a unified and standardised data model and even more a unifying query language. However, standardisation efforts are currently being made in this area within the SQL community, through SQL3 ISO/ANSI [16], and in the OMG community, through OQL ODMG [17]. A few commercial products do currently offer object-oriented query facilities like for instance, OpenODB, UniSQL, and Illustra. Likewise, our AMOS OR DBMS research prototype also include object-oriented query facilities through AMOSQL. The current trend in this area is to combine relational and object-oriented database technologies to achieve object-relational query capabilities, i.e. the powerful query capabilities of relational query languages extended for object-oriented data models, Kim [90].

3.8.1 Relational algebra and relational calculus

It is common to characterise query languages as being *procedural* or *nonprocedural* (also *declarative*). With a procedural language you specify how to compute the expected result, whereas by means of a nonprocedural language you specify the desired result without any computation strategy. Procedural and nonprocedural query languages can be considered as two basic language forms. Most commercial database languages include both nonprocedural and procedural parts as in the case of SQL that include a nonprocedural query language.

Two formal query languages, relational algebra and relational calculus, have been defined for the relational data model.

Relational algebra is a procedural query language that defines a collection of operations for manipulating entire relations. These operations take relations as input and produce new relations as output. There are operations specifically defined for relations as well as set operations from set theory. Examples of the former include `select`, `project`, and join. As illustrated in Figure 11, the select operator extracts tuples from a relation whereas the project operation extracts attributes. The join operation combines related tuples, from two relations, that fulfil some join condition into single tuples in a new relation. There also exist several different variants of join operators. Set operations include union, intersection, difference, and cartesian product. It has been shown that a basic set of relational operators, consisting of the select, project, union, difference, and cartesian product, is complete in the sense that they can be used to form the other relational operators.

Queries in relational algebra are formed by combining operators into an ordered sequence. In the example below the start_point and its position is projected from the resulting table of selecting the lines with line_name "L10" from the straight_line table.

```
PROJECT start_point, position(SELECT line_name =
                              'L10' (straight_line))
```

Relational calculus is, in contrast to relational algebra, a nonprocedural query language. The relational calculus is a formal language based on predicate calculus. Furthermore, their is one variant of relational calculus, termed *tuple relational calculus*, where variables are associated with tuples. There is also a second variant called *domain relational calculus* where variables are associated with attribute values.

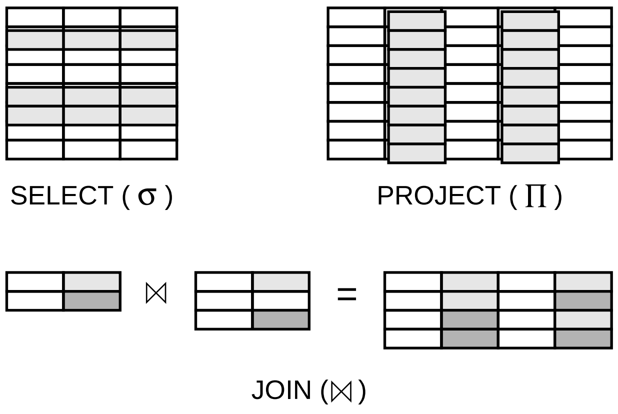


Figure 11. *Illustration of the results of applying three basic relational algebra, operators, SELECT, PROJECT and JOIN, on relational tables.*

For example, for the tuple relational calculus a simple query expression have the form:

```
{tvar | COND(tvar)}
```

where `tvar` is the set of all tuples that satisfy the conditional expression `COND(tvar)`. Simplified, a conditional expression can consist of atomic expressions including tuple ranges and comparison expressions. Atomic expressions can be combined by logical connectives to formulas that can further include quantifiers.

The previous example for relational algebra would have the following form as a query in tuple relational calculus:

```
{t.start_point,t.position | straight_line(t) and
                             t.line_name = 'L10'}
```

The domain relational calculus has a similar structure and it is shown that the expressiveness of domain and tuple relational calculi are the same. It has further been shown that the expressiveness of relational calculus and relational algebra are equivalent. Query languages that have the same expressiveness as relational calculus are called *relationally complete* query languages.

Furthermore, there are other relational data retrievals that can not be expressed by means of the basic set of relational operators. These include aggregation, recursive closure, and arithmetic operations together with specialised variants of the basic operations, such as various join methods.

Most commercial query languages for the relational model include elements of both relational algebra and calculus as exemplified by SQL. Other examples include the query language QUEL that is based on tuple relational calculus and the graphical query language QBE, based on domain relational calculus. Commercial relational query languages usually have relationally complete query capabilities and they also include common extended operations. Their capabilities are extended as new versions appear and their more advanced query facilities will vary between products and versions.

This section has only treated retrieval capabilities of query languages but commercial query languages also include other features covering operations for data definition, updates, and others.

3.8.2 The SQL language

SQL has become the standard query language for R DBMSs. It was first developed by IBM and in 1986 for the American National Standards Institute (ANSI) and in early 1987 for the International Standards Organisation (ISO) an SQL standard was published, Melton and Simon [82]. A revised and expanded version, known as SQL-92 (also SQL2), was published in 1992. There is further ongoing work on the next generation of the SQL standard, SQL3, that aims at including support for object-orientation.

The SQL language is a general query language that copes with data definition, interactive data manipulation, embedded data manipulation, view definition, authorisation, integrity, and transaction control.

The data definition part includes operations for creating, altering, and removing tables, i.e. relations in SQL. In addition to specifying the relations in the database, the DDL includes commands for defining: schemas explicitly, value domains, indexes, security and authorisation, integrity constraints, and physical storage structures of relations.

Updates on relations in SQL consist of the `insert`, `update`, and `delete` commands for inserting, modifying values of, and deleting tuples in a relation.

Retrieval expressions in SQL are formed by means of the `select` statement and its basic structure consist of the `SELECT`, `FROM` and `WHERE` clauses with the following syntax:

```
select    <attribute list>
from      <table list>
where     <condition>
```

where

- The `select` clause corresponds to the projection operator of relational algebra and the `<attribute list>` is the list of attributes that should be retrieved in the query.
- The `from` clause corresponds to the cartesian product operation of relational algebra, and where the `<table list>` is the tables that should be involved in the retrieval
- The `where` clause corresponds to the selection operator of relational algebra where the `<condition>` is a predicate that constrains the attributes in the `from` clause.

Again, the former example would be expressed in SQL as:

```
select    t.start_point,t.position
from      straight_line t
where     t.line_name = ‘L10’
```

This was only a brief presentation of a subset of all the functionality of the SQL language. A detailed description of the current SQL2 standard is provided in Melton and Simon [82]. There is further an SQL3 standard proposal, Melton [16], in progress that suggests major extensions to be made to the SQL language to support object-oriented methodology.

3.8.3 Object-oriented query languages

Object-oriented query languages are based on an object data model in comparison to query languages for R DBMS, which usually means SQL, that are based on the relational data model. Hence, in contrast to relational query languages that operate on tables, object-oriented query languages are designed to operate on sets of objects.

The motivation is to provide query language capabilities for the object data model that are at least as powerful as the relationally completeness of relational query languages. OO modelling is considered to be richer and more suitable for several kinds of applications, including scientific and engineering applications, by providing facilities such as richer data structures, and user-defined types and operations. For instance, data types for numerical collections are usually supported. Furthermore, the object-oriented model reduces the impedance mismatch between the DBMS and the application that exists for the relational data model. Hence, the combination of an object data model with query facilities provides both the advantages of OO modelling and of R DBMSs, such as declarative data access and query optimization.

There is currently no uniform and standardised specification for an object data model. However, there are two emerging standards [116], ODMG-93 [17], and SQL3 [16], that define their own object data model in combination with query facilities. Certain efforts are also made to try to merge these standards. While ODMG-93, including its query language OQL, is more related to OO programming languages, such as C++ and Smalltalk, SQL3 is built on extensions to the SQL language. The final outcome of these standardisation efforts is not settled, and no detailed presentation of their capabilities will be presented here. It is worth noting that the query language of the O_2 DBMS [117] is closely related to the OQL language whereas the query language of Illustra, described in [6], is built on an earlier specification of SQL3.

The AMOSQL query language, described in Section 4.3, will here exemplify the capabilities of an OO query language. It should be noted that AMOSQL originates from the OSQL language [15] and includes some unique features not provided in the suggested standards. These include overloaded multi-directional functions that have been used extensively in this work. Overloading of functions on multiple arguments is included in SQL3 but not in ODMG-93. Further, the extensibility of the AMOSQL query processor that handles optimization of foreign functions is an important property when modelling applications-specific operations. This capability is not covered in any of the standardisations, but is actually provided by Illustra, and emphasised as an important mechanism.

4 THE AMOS DBMS AND THE AMOSQL LANGUAGE

AMOS (Active Mediators Object System) [8] [9] is a research DBMS prototype and conforms to systems classified as *object-relational (OR)* DBMSs. An AMOS prototype has been developed [9], built on substantial developments of the WS-IRiS main-memory OO DBMS engine [103]. WS-IRiS was developed at Hewlett-Packard Laboratories, Palo Alto, CA, and is a derivative of Iris [118]. AMOS is further a main-memory DBMS and assumes that the entire database is contained in main-memory. The AMOS architecture is tailored for main-memory usage to provide competitive performance which is confirmed in [103]. The database can be saved to, and restored from, disk to provide persistency to secondary storage. Optionally, a transactional logging system supports logging and recoverability of committed database transactions.

The "object-relational" term presupposes the existence of a relationally complete query language. AMOS includes the AMOSQL query language that provides a declarative query interface for defining, populating and manipulating the database. AMOSQL is an extensible and object-oriented query language that is more than relationally complete. AMOSQL is a derivative of OSQL [15] which is a functional language, originating from DAPLEX [119]. The AMOSQL query language is further influenced by standardisation efforts like SQL3 [16] and OQL [17].

AMOSQL and AMOS allow extensibility at the query language level, the query processing level, and at the storage management level. These capabilities are important for accommodating tuned representations and operations required by demanding applications. Extensions to the system can be implemented in external programming languages like C or LISP (or languages callable by these).

AMOS can be operated as a single-user system, or as a multi-user system using client-server network communication. In addition prototypes of graphical interfaces to AMOS have been developed that include web-server capabilities to be able to communicate with web-browsers.

The next section presents the mediator approach that shows how AMOS servers can be used as a mediating layer between applications and data sources in, for example, an EIS environment. The idea of domain-specific mediators is explained. This is followed by a presentation of the AMOS architecture including the different subsystems that form AMOS and their interaction. A short introduction to the AMOSQL language is then included and, finally, this chapter is completed with a more detailed presentation of the possibilities of extending and interfacing AMOS.

4.1 THE MEDIATOR APPROACH

Future computer-supported engineering environments will consist of large numbers of workstations connected with fast communication networks. Each workstation will have powerful computing and storage capacities to store, maintain, and do computations over local engineering data and knowledge bases, or information bases. The *mediator* approach, Wiederhold et al. [12], Wiederhold [13], and Risch and Wiederhold [14], introduces an intermediate level of software between databases and their use in applications and by users. The purpose of a mediator is to query, monitor, transform, combine, and locate desired information between a, possibly heterogeneous, set of applications and data sources. An external data source can be a conventional DBMS such as an R DBMS or an OO DBMS, other mediators (other AMOS servers in our case), data files for specific exchange file formats, as well as data obtained by executing some program. Figure 12 shows a possible configuration of a mediator system. Since mediators “understand” application terminology as well as database terminology, they can combine, and take advantage of, both high-level domain-specific data interaction and efficient data management.

Several different types of AMOS mediators can be identified including domain models, monitors, integrators, translators, and locators:

- *Domain models* are complete or partial models of application domains and this type of mediator is of main importance to this work. This can include the complete conceptualisation of the domain with both concept structures and operators. The extensibility of AMOSQL allows the design of domain models that represent application-oriented models and operators, i.e. FEA models in our case. This allows for knowledge, now hidden within application programs as local data structures, to be extracted and stored in AMOS modules as domain-specific models and operators. The benefits of using domain models include easier access and management through a query language, better data description (as schemas), and other benefits provided by DBMSs such as transaction capabilities and ad hoc query processing. The query processing must be as efficient as customized main-memory data structure representations to allow the use of local embedded domain models linked into applications without substantial loss of efficiency. Domain models often need to be able to represent specialized data structures for the intended class of applications. An initial study on how the AMOSQL language can be used to model and manage domain models for product data has been reported in Orsborn [120] [121]. Initial results from the use of AMOSQL to model FEA domain models have been presented in Orsborn [7]. Domain models for FEA are further discussed in Section 5.3. It is possible to use both programmed procedures and high-level query languages for accessing domain models. Combining general query-language constructs with domain-related representations provides a more problem-oriented communication. Investigations show that this approach to data management is more effective compared to the use of conventional programming languages [76] [122]. The combination of programming and query languages and their pros and cons for data management are further discussed in Cattell [85].

- *Monitors* are mediators that manage the detection of significant changes in data (or in relations among data) that could originate from some sensory data sources or other updates of data. Active mediators can respond to changes and take action or notify applications or mediators. The treatment of monitors in AMOS by providing active rules is discussed in Risch and Sköld [108], Sköld [109], and Sköld and Risch [110].

- *Integrators* and *translators*. Integrators combine data from several data sources or other mediators, to form a uniform view of combined data. In the integration process it might be necessary to first translate data by a translator to obtain a uniform representation format. By using this type of mediator, applications can be designed for one uniform representation and there is no need to maintain multiple data models. Integration and translation in the AMOS environment is treated in Fahl [123] and Fahl and Risch [124].

- *Locators*, a generalised form of *name servers*, are servers for locating where information is stored among data sources or other mediators in a distributed environment. An important part in locating information is the ability to treat global queries, i.e. queries that span several databases. Multidatabase queries for distributed AMOS systems are treated in Werner [93].

As illustrated in Figure 12, a specific mediating system might require a composition of several, but maybe not all, types of mediators. Hence, the mediator architecture should allow a combination of different mediators into a mediator system. It would then be possible to design dedicated mediators for specific tasks, e.g. for accessing an R DBMS, for import and export to and from data exchange files, for instance. This type of mediating system can then be accessed by other mediating systems in the network. Further, the example in Figure 12 illustrates different alternatives for applications to access data in external data sources in the mediator system. In Application 1, only one external data source is relevant and there is no need to use an integrator to access data. The translator provides data in a format that can be directly accessed through the embedded AMOS in the application. The next case, Application 2, includes queries that span over two data sources that require an integration of data. The final case is a combination of the two previous situations. Some application queries span several data sources and need integration whereas other queries only regard data in a third data source that can be accessed directly through the translator. Each application can further store local data in its embedded database.

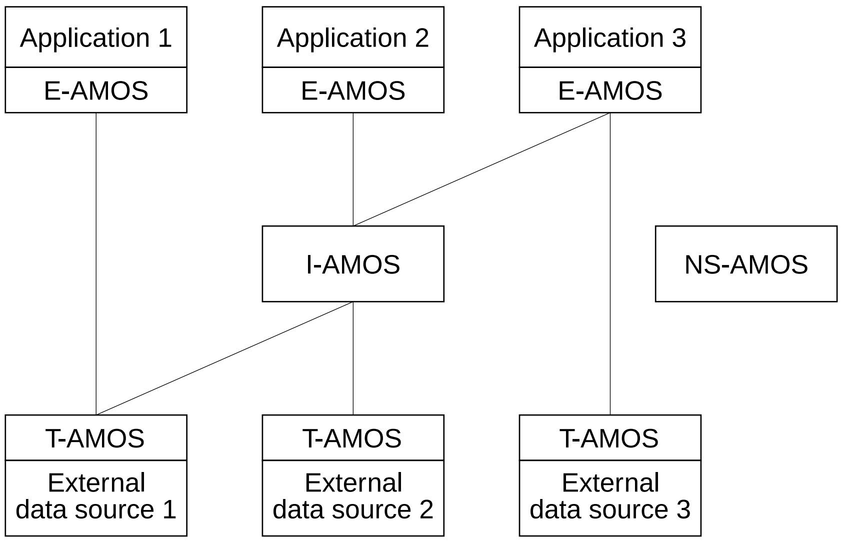


Figure 12. *A mediator architecture that illustrates how a combination of different types of mediators (AMOS DBMSs) is used to mediate data among applications and data sources. Each application is equipped with an embedded DBMS.*

The domain models are not indicated in Figure 12, but can constitute parts of other mediators. That is, an embedded or an integrator AMOS can include domain-oriented meta-data in their schemas. Meta-data for each application domain that should be handled by a mediator must be available to its schema. Likewise, mediators that should handle active behaviour must be configured with monitoring capabilities.

4.2 THE AMOS ARCHITECTURE

The AMOS design intends to provide a lightweight[1] and open DBMS architecture that should permit an easy combination and integration with other applications. It should further facilitate tailoring and extension of the DBMS to suit the needs of demanding applications as found in the engineering area. AMOS is intended to work as a mediating software layer among applications and data sources for locating, storing, retrieving, exchanging, transforming, and monitoring data.

As illustrated in Figure 13, AMOS can be an embedded database within an application by directly linking it to the application at compile time. The application and the DBMS will be executing in the same computer process and be sharing its address space. In addition, the DBMS can be used in a client-server environment where the applications and the DBMS have their own computer processes, also shown in Figure 13. The client-server communication is based on sockets and remote-procedure calls. It is also possible to define domain-specific packages of specialised data structures and operators, and integrate them with AMOS. AMOS has the ability to seamlessly define and call foreign functions (implemented in, or callable from, C or LISP) through the foreign data source (FDS) interface. More about foreign functions and extensibility of AMOS is presented in the rest of this section and in sections Section 4.4 and Section 5.2.

The AMOS kernel consists of several subsystems that are responsible for different tasks. The main subsystems are illustrated in Figure 14 and include:

- The communication with AMOS from external applications is made through an external call-level interface. There are currently interface procedures, that can be called from application programs, supported for the C and LISP languages. Further, the interface support both fast-path communication with AMOS and communication through embedded AMOSQL statements within applications. The fast-path interface is a procedural call-level interface that accesses precompiled database functions directly, bypassing the parsing and optimization steps. Type checking on AMOSQL function arguments is optional and late bound function invocations are permitted. In the embedded approach, AMOSQL statements are embedded in applications and passed as strings to an evaluation function that activates the AMOSQL parser. Both approaches can be mixed within applications. The same external interface supports both communication with embedded databases and client-server communication as illustrated in Figure 13. AMOS also has a foreign data source (FDS) interface for external communication with other applications, tailored application packages, or other AMOS databases as well. See subsequent item on the FDS for more details on integrating external data sources.
- A *command interpreter* that is responsible for scanning and parsing of AMOSQL expressions. The parser passes requests to an appropriate subsystem such as the

1. The AMOS footprint is about 800 KB of code and 800 KB of meta data on HP7xx series workstations.

schema manager, rule processor or the query optimizer. It further dispatches commands to the transaction manager for committing transactions, saving databases and connecting to databases.

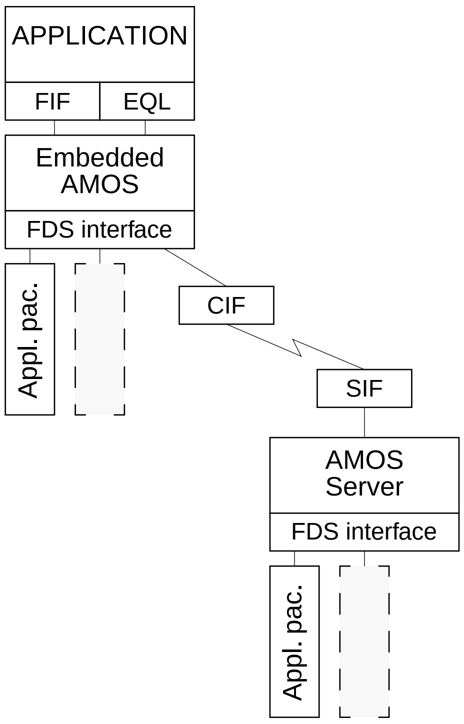


Figure 13. *A possible client-server architecture for working with applications coupled to AMOS. The fast-path interface (FIF) and the embedded AMOSQL (EQL) can be used to communicate with the DBMS by applications with an embedded DBMS. AMOS also includes a foreign data source (FDS) interface for integrating foreign data and operator representations.*

- The *schema manager* controls all schema operations. This includes creating and deleting types, functions, and rules.
- There is also a *rule manager* that handles rules in the database. Rules in AMOS are of condition-action type where conditions stated as general AMOSQL queries can trigger actions to be performed, expressed as general AMOSQL procedural statements. Rule management includes issues such as creation, deletion, activation, de-

activation, monitoring and execution of database rules. Rules in AMOS are treated in more detail in Risch and Sköld [108], Sköld [109], and Sköld and Risch [110].

- The *foreign data source* (FDS) interface of AMOS admits integration of foreign data structures and operators. Foreign operators are defined and implemented as *multi-directional foreign functions* with overloading on all arguments, Litwin and Risch [103], Flodin and Risch [86], Flodin [87], and Flodin et al. [19]. In AMOSQL, one can define database operations as foreign functions that are implemented in some external programming language like C or LISP. An AMOSQL foreign function is seamlessly integrated within the query language. It is further possible to define and register new foreign data representations that will be accessible within the database. Extensibility and foreign functions of AMOS are further discussed and exemplified in sections Section 4.4 and Section 5.2.

- The *optimizer* is responsible for transforming ad hoc queries, update statements, functions, and procedures into tractable execution plans using query optimization and compilation techniques. This process involves the application of transformation rules and heuristic cost-based query optimization techniques that produce executable and efficient query plans. By the definition of execution costs for foreign functions (default costs are also provided), the optimizer is able to optimize expressions that include foreign functions. Query optimization in AMOS and the management of foreign predicates are treated in detail in Litwin and Risch [103], Flodin and Risch [86], and Flodin [87].

- The *execution plan interpreter* handles the processing and execution of optimized expressions that are represented in an intermediate ObjectLog language, Litwin and Risch [103]. The execution plan interpreter is, for instance, responsible for dispatching calls to the FDS interface.

- The *logical object manager* administers operations on database objects including OID handling, creation and deletion of objects, and updating of stored functions. Updates can imply inserting, updating, or removing data in functions whose extents are stored in the database. An update operation causes the creation of an event that is passed to the event manager in the physical object manager. Operations on this level are transactional and are optionally logged. The logical object manager has a fast path entry for calling preoptimized AMOSQL functions from the call-level interface.

- The *physical object manager* includes parts for managing all physical operations on user objects (i.e. instances of user-specified types), system objects (strings, integers, reals, lists, arrays, vectors, atoms, hash tables, etc.), and event objects (objects representing database transactions). Examples of operations are allocation, deallocation, and access operations. Foreign functions can manipulate the physical object manager, e.g. to allocate and update user-defined internal storage structures.

- Memory operations are controlled by the *memory manager* that automatically allocates and deallocates memory, and reclaims memory by garbage collection. Thus, memory management is implicitly controlled by the DBMS, relieving the programmer from explicit memory handling.

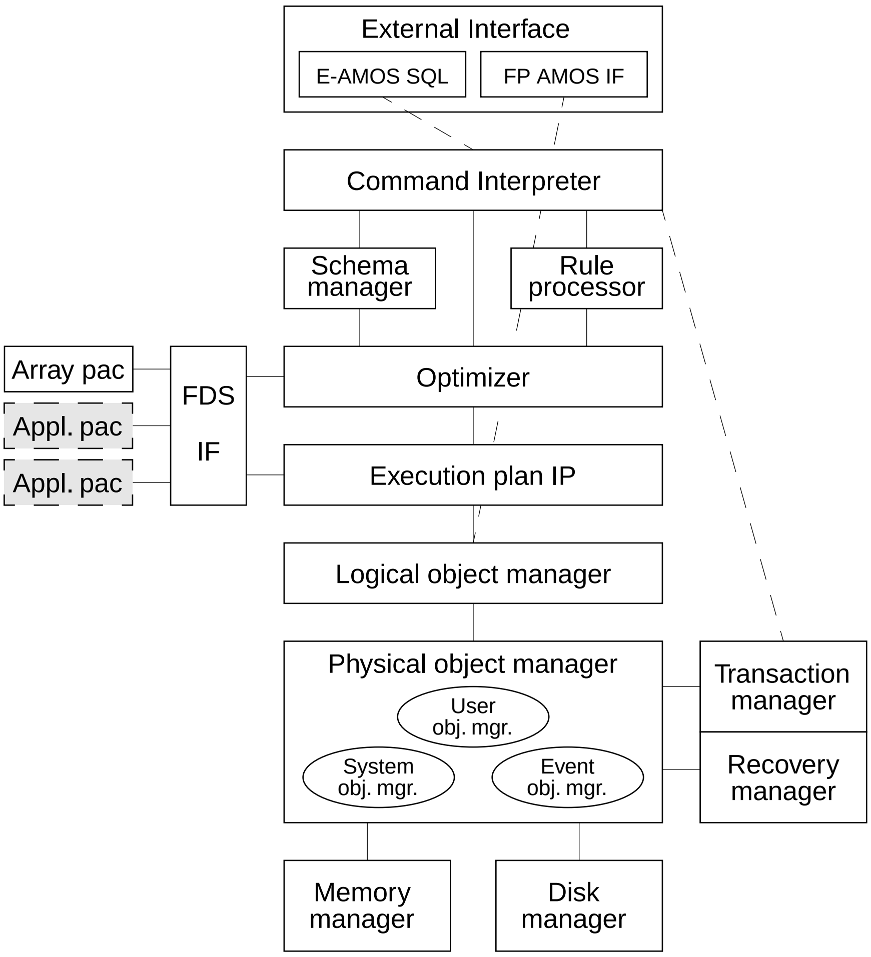


Figure 14. *A schematic view of the AMOS architecture where boxes represent main functions.*

- Since AMOS presupposes that the database resides in main-memory, the *disk manager* is more primitive in comparison to disk-based DBMSs. It mainly handles flushing of database images between main-memory and disk for initiation, connection, or saving of databases.
- The *transaction manager* controls all transactions to the database by keeping a log

of all database operations so that transactions can be undone or redone to guarantee database consistency.

- Optionally, a *recovery manager*, Karlsson [125], can be activated to ensure database persistency. The recovery manager is responsible for automatically maintaining persistency of a database that is exposed to transactions. Persistency and backup, with respect to secondary storage, is in AMOS accomplished by flushing the log-file to disk for each transaction and saving the database to disk periodically. By logging transactions to disk, the database can be recovered after a main-memory crash.

The architecture of AMOS permits extensions to be made on all levels that were described in Section 3.4 and the extensibility of AMOS will be further discussed in Section 4.4.

4.3 THE AMOSQL LANGUAGE

AMOSQL is a functional language with object-oriented extensions and with an expressiveness beyond relational completeness. Its basic capabilities include constructs for database schema definition and evolution, database population and updates, and database queries in terms of the basic data model that includes objects, types, and functions. It further supports logical operators, arithmetic operators, aggregation operators, nested subqueries, disjunctive queries, quantification, transitive closures, recursion, multi-database queries, Werner [93], and active rules, Risch and Sköld [108], Sköld [109], and Sköld and Risch [110]. Other main features are capabilities to handle overloaded and multi-directional functions in combination with late binding, Flodin [87], Flodin and Risch [86], and Flodin et al. [19]. These facilities can be handled for both regular and foreign AMOSQL functions.

AMOSQL provides a declarative query language interface to the database. The declarative nature requires optimization of queries before execution can take place to accomplish efficient execution strategies. AMOS currently supports three different optimization techniques that cover classical exhaustive search and newer techniques based on heuristics, Litwin and Risch [103] and Näs [126]. By defining additional cost formulas expressed as AMOSQL functions, the optimizer can be extended to handle costs for, and optimize queries including, foreign functions.

Extensibility of AMOSQL is further treated in Section 4.4 and for a more detailed presentation of the AMOSQL language, the reader is directed to Flodin et al. [9].

4.3.1 Objects, types, and functions

The data model consists of the basic constructs *objects*, *types*, and *functions* as illustrated in Figure 15. Concepts in an application domain are represented as *objects*. There are two types of objects in AMOS. *Literal objects,* such as boolean, character string, integer, real, etc., are self identifying. The other type are known as called *surrogate objects* that have unique object identifiers. Surrogate objects represent physical or abstract and external or internal concepts, e.g. mechanical components and assemblies such as a skin panel or a wing in an aircraft design, finite elements, geometrical elements. System-specific objects, e.g. types and functions, are also treated as surrogate objects.

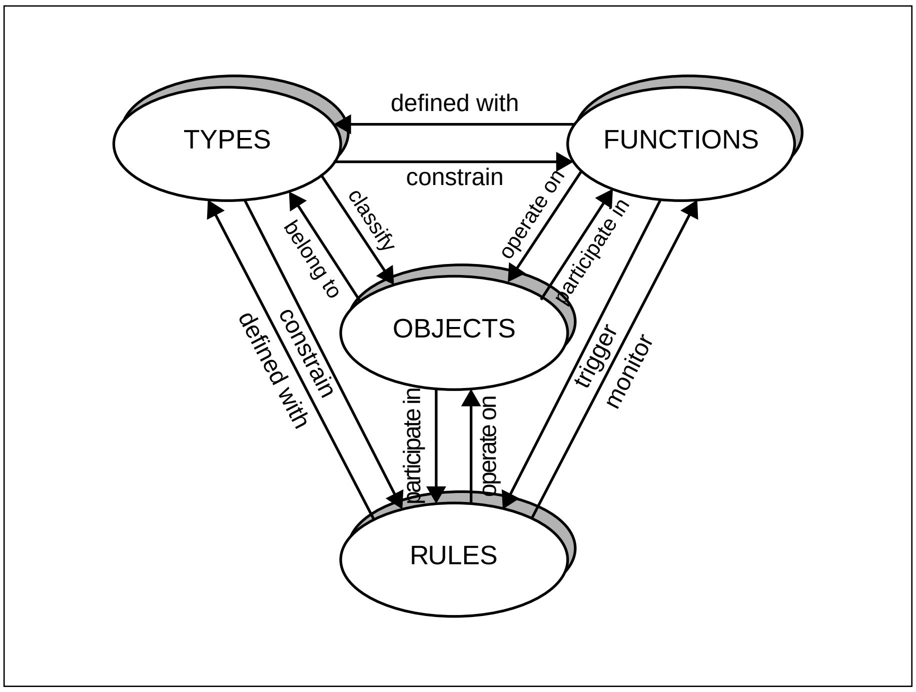


Figure 15. *The basic constructs and relations of the AMOS data model.*

Types are used to structure objects according to their functional characteristics, in other words it is possible to structure objects into types. Types are in themselves related in a type hierarchy of subtypes and supertypes. Subtypes inherit functions from supertypes and can have multiple supertypes. In addition, functions can be overloaded on different subtypes (i.e. having different implementation for different types). The current type taxonomy of AMOS is presented in Figure 16.

Functions are defined on types, and are used to represent attributes of, relationships among, and operations on objects. Examples of functions for these different categories might be diameter, distance, and move_point. It is possible to define functions as *stored, derived, procedure* or *foreign*. A stored function has its extension explicitly stored in the database, whereas a derived, procedure, or a foreign function has its extension defined in an AMOSQL query, an AMOSQL procedure, or a function in an external language. Furthermore, functions can be defined as one- or many-valued and are invertible when possible. Stored and some derived functions can be explicitly updated using update semantics but other functions need special treatment for updates. Advanced OO query-languages also provide *object views* [127] capabilities to support data independence. In AMOSQL, views are supported through derived functions. Functions provide an associative access to objects and are uniformly invoked independently of whether they represent stored or derived data. This makes it possible to change the underlying physical object representation without altering the access queries. Since functions can be optimized they support a higher level of data independence than methods.

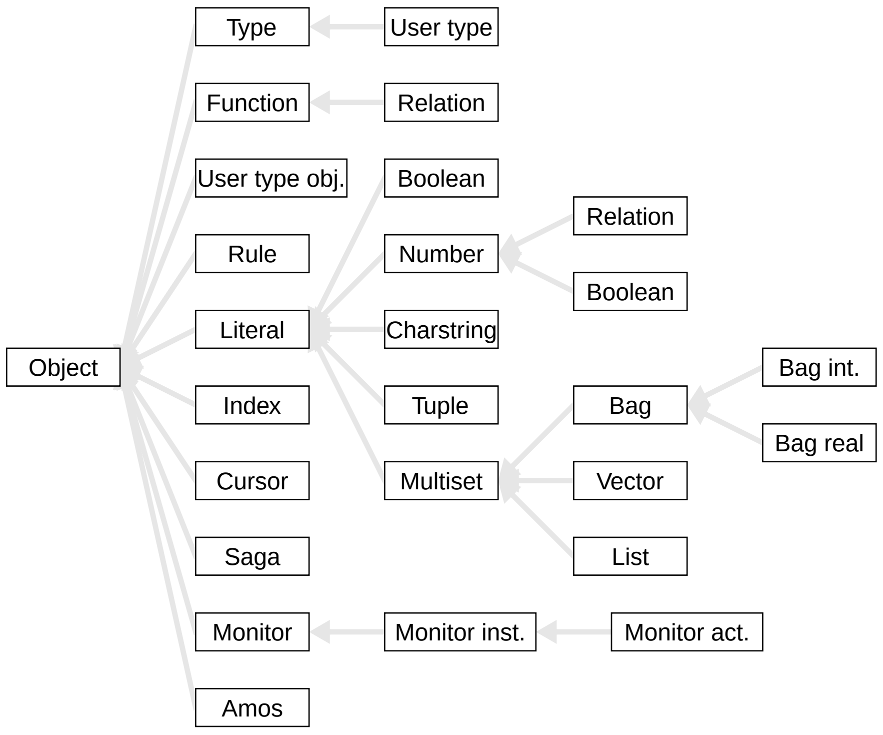


Figure 16. *The current type taxonomy of AMOS with text-boxes indicating entities and arrows indicating is-a relationships.*

Functions can further be *overloaded*, i.e. functions defined for different combinations of argument types can share the same name. This is also referred to as a special form of *polymorphism*. The selection of the correct implementation of an overloaded function is made at function invocation based on the actual argument types. A variant of an overloaded function, i.e. a specific implementation, is called a *resolvent*. AMOS supports overloading for all four basic types of functions and automatically handles the selection of early binding (compile time) and late binding (run time) of functions. Overloading of functions on the result types is not currently supported.

In addition, AMOS supports *multi-directional functions*, i.e. a function can be applied for different binding patterns. A *binding pattern* defines which of the arguments and the results that are bound, and which that are unbound, in a function invocation. In this way, functions can be applied for several “directions”, or binding patterns, making the function similar to a relation. For stored, derived, and procedural functions, this multi-directional capability is automatically supported where applicable. In the case of foreign functions, the user must explicitly define the applicable variants of the multi-directional function.

A thorough presentation of the AMOS treatment of overloading and multi-directional functions is given in [19] [86] [87].

4.3.2 AMOSQL data management

AMOSQL provides statement constructs for typical database tasks, including data definition, population, updates, querying, flow control, and session and transaction control. Data schemas can be defined, modified, and deleted by means of AMOSQL statements. The definition of types, functions, and objects is performed through a `create` statement. For example, types may be defined by a `create type` statement as:

```
create type named_object(name charstring);
create type fea_object subtype of named_object;
create type element subtype of fea_object;
create type node subtype of fea_object;
```

where a stored function, `name`, is defined as a character string within the parentheses of the `named_object` type. A new type becomes an immediate subtype of all supertypes provided in the `subtype` clause, or if no supertypes are specified, it becomes an immediate subtype of the system type `UserTypeObject`.

Functions can also be defined separately from the types by a `create function` statement, exemplified by the `nodes` function that relates elements to nodes:

```
create function nodes(element e1) -> bag of node n as stored;
```

A database is populated with objects with a `create` *`type`* statement with or without initialisation of functions, and where *`type`* stands for the specific type to be instantiated. For example, some nodes and elements can be created by the following statements[1]:

```
create node (name) :n1 (“n1”),
                   :n2 (“n2”),
                   ...
                   :n16 (“n16”);

create element (name, nodes)
                        :e1 (“e1”, bag(:n1, :n2, :n6, :n5)),
                        :e2 (“e2”, bag(:n2, :n3, :n7, :n6)),
                        ...
                        :e9 (“e9”, bag(:n11, :n12, :n16, :n15));
```

Derived functions are defined in a similar manner as stored functions with a single AMOSQL-query as the function body. An example of a derived function is presented as the `topology` function below[1]:

```
create function topology(element e1) -> element e2 as
       select distinct e2
              for each element e2
                 where nodes(e1) = nodes(e2) and
                       e1 != e2;
```

The topology defines how elements are related to each other. When the `topology` function is accessed, it derives the elements `e2` that have some common node with element `e1`, i.e. the elements that are connected to a given element. An example shows the topology for elements 1 and 5, respectively[2].

```
name(topology(:e1));
“e2” “e4” “e5”

name(topology(:e5));
“e1” “e2” “e4” “e6” “e8” “e9” “e7” “e3”
```

Querying a database for objects having specified properties is made using a `select` statement. For instance the `nodes` of `:e1` in the example earlier can be retrieved by the following query:

```
select name(nodes(:e1));
“n1” “n6” “n5” “n2”
```

Functions are also invertible (not always) and it is therefore possible to use the `nodes`

1. Variables preceded by colon, such as :n1, are global interface variables used by AMOSQL to hold query results temporarily during a session and to share values with foreign languages.
1. The “=” operator in this case compares each element in the bags.
2. Functions can be composed in accordance with the DAPLEX semantics [119]. This means that if the result of an inner function is multiple-valued, the subsequent outer function is applied to each value.

function in the opposite direction which can be expressed as:

```
select name(e) for each element e where nodes(e) = :n1;
"e1"
```

Deletion of types, functions, and objects is performed through a `delete` statement as:

```
delete type element;
delete function nodes;
delete :e1;
```

In addition to database population by object creation and attribute assignments it is possible to use *function update statements* `set`, `add`, and `remove`, and *type update statements* `add` and `remove`. Examples of update statements for functions are:

```
set nodes(:e1) = bag(:n1, :n2, :n6);
add nodes(:e1) = :n5;
remove nodes(:e1) = :n2;
```

A more complete presentation of data management capabilities in AMOS and AMOSQL is presented in [9].

4.4 EMBEDDING, INTERFACING AND EXTENDING AMOS

The two basic alternatives for connecting applications to AMOS are either through a *tight* or a *loose* connection.

In the tight connection, or *embedded DBMS connection*, AMOS is directly linked together with a C-based application program. Since this means that the application and AMOS are executing in the same address space, it provides the fastest connection possible. By using a *driver program* in C that initialises AMOS and catches AMOS errors, the DBMS can be linked to the application as a C-library. A possible disadvantage with this approach is that execution errors in the application may cause AMOS to crash.

For the loose connection, or the *client-server connection*, the application can work as a client to an AMOS server. A client-server connection permits that several applications access the same AMOS server concurrently. In this situation, the application and AMOS are executing in different Unix processes. This approach makes the AMOS server more resistant to execution errors in the application. If an application crashes, it will not afflict the AMOS server. The main disadvantage in this approach is the overhead of the inter-process communication. In comparison to the tight connection, the access time can be several order of magnitudes higher in this loose connection.

For both these types of connections there are two possibilities to communicate with

AMOS from the host language of the application; either through the *embedded query interface* or through the *fast-path interface*. Currently, the host language should be C (or being able to call C), since the interface routines are available in C.

In a tight connection, the embedded query interface, AMOSQL statements can be executed from the host language by calling an AMOSQL execution procedure that takes strings of AMOSQL statements as input. After evaluating the AMOSQL statement, it returns the value of the evaluation as the result. Carrying out the evaluation means that the parser and the query processor must be activated for interpreting and executing the statement.

This can be avoided by using the fast-path interface through which calls to predefined and preoptimized AMOSQL functions can be made directly, thereby evading the parsing and query processing step. It should be noted that using the fast-path, instead of the embedded query interface, is significantly faster. According to [9], the difference is up to two orders of magnitude. The normal database-access from the application should use this technique of defining AMOSQL functions for database-related application operations and later invoke them from the application using the fast-path interface.

In addition to the procedure for using the embedded query and fast-path interface, respectively, the inter-process communication must be activated when the client-server connection should be used.

For transferring data between the host language and the database AMOS uses *object handles*, i.e. logical references to C data structures stored in the database. An operation, termed *dereferencing*, can convert the object handle into the C pointer that refers to the corresponding data record. The interface library provides routines for managing data referenced by object handles, such as primitives for declaring, assigning, deassigning, and dereferencing object handles.

Furthermore, AMOS includes an embedded Lisp interpreter that provides similar capabilities (and sometimes simpler) for exchanging Lisp data with the database as was described for C. There also exist interface routines between C and Lisp.

According to the previous Section 3.2, there were three categories of extensibility that could be identified for an extensible DBMS. These were query language, query processor, and storage manager extensions. AMOS is extensible on all these levels.

At the query language level, the user can define application-specific types and operations using AMOSQL. How users can create new types in AMOSQL was described in Section 4.3, that also included examples on how to define operations as AMOSQL functions.

It is further possible to seamlessly extend AMOSQL with new database operations by defining AMOSQL functions as foreign and using a conventional programming language (such as C or Lisp) for their implementation. This is done by means of the *foreign function interface* that allows AMOSQL to communicate with external programming languages.

AMOSQL *foreign functions* can be *uni-directional* or *multi-directional*. A uni-directional function is a normal function with the simple intention to compute the result (one or several) given the values of its arguments. The following example shows how to define a uni-directional function `abs`, that computes the absolute value of a number using a foreign Lisp function:

```
create function abs(number n1) -> number n2 as
                 foreign abs.number.number;
```

where `abs.number.number` is the name of the Lisp function that implements the `abs` AMOSQL function. The `abs.number.number` implementation only includes a call to the Lisp `abs` function:

```
(define abs.number.number (obj osql:n1 osql:n2)
   (osql-result osql:n1 (abs osql:n1)))
```

Like AMOSQL functions, foreign functions can be multi-directional. A multi-directional function is defined, and can be applied, for different binding conditions. The binding conditions state the admissible combinations of bound and unbound (free) arguments and result variables for a certain function and are defined in a *binding pattern*, i.e. a list of - (bound) and + (free) symbols. For uni-directional functions, the arguments are always bound and the results are unbound that state the (- +) binding pattern for the `abs` function. The reasons for having this multi-directional capability include a reduction of the number of operator signatures that are necessary for a specific amount of functionality and that the query optimization is enhanced. By reducing the number of operations while keeping the functionality will further result in simpler domain models and support reusability. To exemplify this capability we show how the minus function can be implemented by inverting the `plus` function[1].

```
create function minus(number n1, number n2) -> number n3 as
             select n3 where n3 + n2 = n1;
```

Here, the definition of the minus AMOSQL function reuses the `plus` multi-directional foreign function and for different possible binding patterns the correct variant of the `plus` function will be applied. The definition of the `plus` function must be made for each applicable binding pattern. The most obvious ones are those that correspond to one

1. The “+” operator is in-fix syntax for the `plus` function. Furthermore, the `minus` and `plus` functions are actually system-provided functions but are here used in a simple example to include most of the interesting ingredients.

free argument or result. This gives us the following three definitions[1]:

```
create function plus(number n1 bound,
                     number n2 bound) -> number n3 free
                 as foreign plus.number.number.number--+;

create function plus(number n1 bound,
                     number n2 free) -> number n3 bound
                 as foreign plus.number.number.number-+-;

create function plus(number n1 free,
                     number n2 bound) -> number n3 bound
                 as foreign plus.number.number.number+--;
```

In similarity with the `abs` function, each `plus` is implemented by a foreign Lisp function. The `plus` (and the `minus`) function can now be applied with either of the binding patterns. For instance, to find the sum and the difference of two numbers would look like:

```
select n3 for each number n3 where plus(:n1,:n2) = n3;

select n3 for each number n3 where minus(:n1,:n2) = n3;
```

Furthermore, again in correspondence with AMOSQL functions, AMOS can handle *overloaded foreign functions* with overloading on all arguments. This makes it possible to define foreign functions for different combinations of argument types and AMOS will be able to select the correct variant at invocation. Examples of the use of overloaded foreign functions are given in Section 5.2.

The query processor must know about the cost of different operations to be able to perform query optimization. For this reason, the query optimizer of AMOS can be extended with *cost hints* for foreign functions, making it possible to optimize queries that include foreign functions. A cost hint currently consists of two values, one that represents the cost of the operation and another that represents the size of the result. Cost hints can be provided as absolute values, by evaluating a cost hint function, or by a default hint if no cost hint is supplied. There are additional possibilities for extending the query processing that is exemplified in Näs, [126], that treats the extension of AMOS with new optimization techniques.

The AMOS storage management system further allows the extension of AMOS with new *storage types*, i.e. different types of physical data records. The storage manager is responsible for allocation and deallocation of physical objects that are referenced through object handles. The foreign function interface provides primitives for defining

1. The syntax used here is adapted to a planned AMOSQL evolution that slightly simplifies the definition of multi-directional functions.

this kind of new data storage types. AMOS maintains a global type table with information about the properties of the built-in physical storage types, such as integer, real, string, object, etc. The introduction of a new storage type is made by expanding the type table with a new entry with information that includes the type name, size, allocation method, deallocation method, and more. In addition, each storage type should have a corresponding C record template that should include type information, a reference counter, and the data part. This capability can be used to define tailored data structures, required in many scientific and engineering applications, that can be made available in AMOSQL.

The AMOS System Manual [128] treats most of the issues discussed in this section in more detail. Various aspects of foreign functions are discussed in Litwin and Risch [103], Flodin and Risch [86], Flodin [87], and Flodin et al. [19].

5 THE FEAMOS APPROACH

This chapter describes the FEAMOS system in more detail, starting with a recapitulation of the motivation for the approach of using database technology as a basis for implementing FEA applications. The next section presents the FEAMOS architecture that shows how the previously presented AMOS architecture is specialised for FEAMOS. The internal architecture shows how the FEAMOS application is designed, which is followed by an illustration of how it fits in a mediator-based EIS environment. After the architectural description, there is a section that describes the design and implementation of the matrix and array data sources that extend AMOS with storage structures and operations suitable for numerical and linear matrix algebra within the database. The subsequent section describes the current FEA domain model that has been developed for the FEA domain. The presentation of this high-level FEA domain modelling is divided into subsections that cover specific parts, including: geometry, mesh, boundary, solution algorithm, result interpretation. The final subsection treats performance issues for the current implementation status of FEAMOS where comparisons are made with TRINITAS.

The FEAMOS system consists of the TRINITAS FEA application equipped with a local embedded AMOS DBMS linked to the application. By using this approach the application gains access to general database capabilities tightly coupled to the application itself. On the external level, this approach facilitates data mediation among applications and data sources in the EIS environment. Furthermore, this provides the application internally with complete DBMS capabilities.

The basic idea in the FEAMOS approach is to use database technology as a basis for developing the FEA application. In our case, the conventional database approach of only storing and accessing data is also extended with the possibility to perform application-specific operations on data within the database. In this context, we use the term computational database technology to provide a vocabulary for referring to database technology that supports applications with a major need for computational support, which is here exemplified by FEA for computational mechanics.

To recapitulate, we have identified certain database technologies that we think are important and suitable as computational database technologies for this task. These key technologies are main-memory resident, extensible, and object-relational database technologies in combination with the domain model concept.

Main-memory DBMSs, such as AMOS, are prerequisites for the embedded database approach and to support processing efficiency comparable to that of programming languages.

Extensibility should be provided for the query language, the query processor and the storage manager. The query language extensibility of AMOSQL provides user-defined types and operations and supports powerful means for flexible domain modelling while letting the DBMS perform data access optimization. AMOSQL also provides extensibility of the query optimizer, which permits the introduction of more complex domain-specific cost models that reflect certain aspects of the application domain. For example, costs for numerical operations or solution accuracy of numerical calculations can be included in the cost models. Thus, the query optimization can influence the choice and tuning of operators in FEA. Furthermore, the AMOS storage manager can be extended with tailored data representations that can be specialised for numerical operations or other application-critical tasks.

The term *object-relational* presupposes the existence of an extensible OO query language. The AMOSQL query language provides the application with a declarative access language to the database. A declarative query language in combination with query optimization supports high-level modelling with powerful capabilities for data independence. In addition, AMOSQL has the unique capability of supporting multi-directional functions with overloading on all arguments that extends the modelling capability and increases the possibility of reuse. Query languages also make it possible to make advanced *ad hoc queries* concerning the contents of the database. This might be demanded by advanced users and is quite useful since it is impossible to foresee the complete information needs.

These database technologies form the basis for the FEAMOS approach to define *domain models* that represent a conceptualisation of the FEA domain and which can take advantage of DBMS functionality for its definition, compilation, and optimization.

5.1 THE FEAMOS ARCHITECTURE

The architecture for the integrated FEAMOS system is illustrated in Figure 17. The application and the database are linked together and communicate through the fast-path interface (FIF) provided by AMOS. This makes it possible to call precompiled and preoptimized database functions. Query expressions can also be sent to AMOS using the embedded query language interface (EQL). Furthermore, through the foreign data source (FDS) interface the DBMS can access application-specific packages that can include specialised data structures and operations.

By this approach the application gains access to general database capabilities tightly coupled to the application itself, providing a storage manager, data model, database schema, database language and processor, transaction processing, and remote access to data sources. Internally, the architecture provides the application with powerful and high-level modelling capabilities through the object-relational query language. This includes object identifiers, subtyping, inheritance, views, overloaded and multi-directional functions, and foreign functions. The modelling capabilities make it possible to design database schemas that possess both physical and logical data and operator independence. Hence, the query language modelling supports and facilitates high-level application modelling that increases flexibility, composability, and reusability of domain conceptualisations.

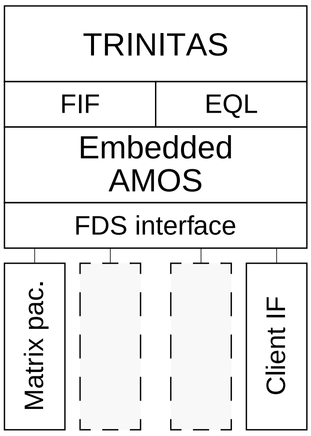


Figure 17. *The FEAMOS architecture with the AMOS DBMS embedded within the TRINITAS application. for working with applications coupled to AMOS. Currently the fast-path interface (FIF) is used but the embedded AMOSQL (EQL) is also available for applications with an embedded DBMS. A matrix package is integrated with the DBMS as a foreign data source (FDS).*

On the external level, this approach supports, among other things, concurrency, interoperability, data exchange and transformation, data and operator sharing, data distribution among applications and data sources in the EIS environment as illustrated in Figure 18. Different AMOS mediators can be combined for locating, translating, and integrating data in various data sources for the applications. Ultimately, the DBMS can decide how and where to execute a query, using query optimization techniques. By providing several mediators with the same application package, as illustrated by the matrix package in Figure 19, it will be possible to decide where the execution should take place.

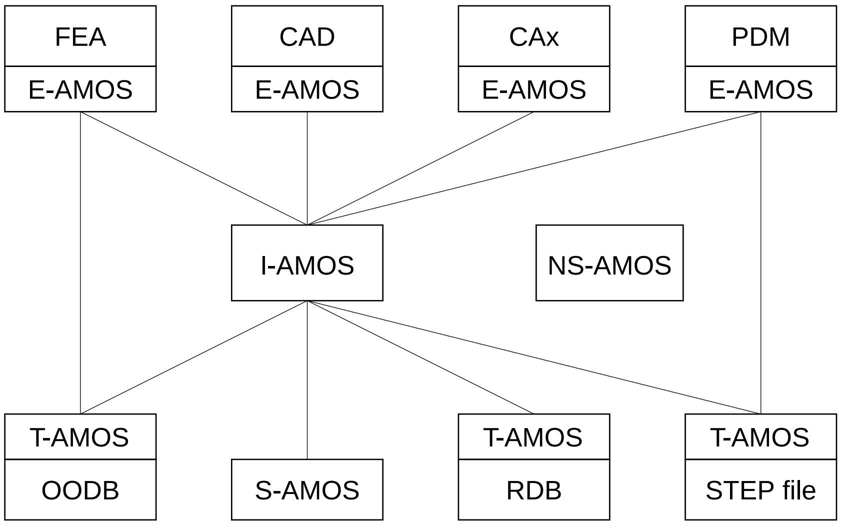


Figure 18. *A mediator-based EIS architecture that illustrates how a combination of different types of mediators (AMOS DBMSs) is used to mediate data among applications and data sources. Each application is equipped with an embedded DBMS.*

Figure 18 shows an example of how an EIS environment could be extended with data management capabilities by taking advantage of the mediator approach. Each application can have its own embedded DBMS (E-AMOS) that can communicate with other mediators. For example, an integrator (I-AMOS) and possible translators (T-AMOS), can support the FEA application to retrieve analysis data that reside in a relational database (RDB) or in a STEP import file. The CAD application can provide geometric data through its own mediator to a database server (S-AMOS). An OO database (OODB) can be used for long-term storage of FEA data. A product data management (PDM) system can be used for administrating product data in the EIS system. The name server (NS-AMOS) is used to locate where data is stored. The domain models that cover the different applications are parts of the global schema of the mediator system.

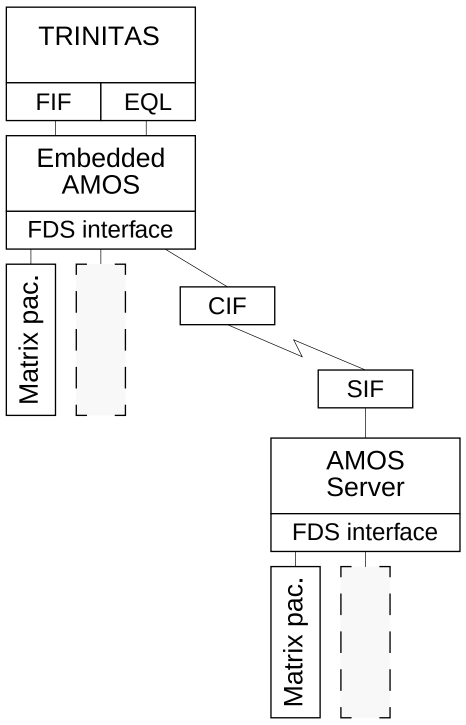


Figure 19. *A possible client-server architecture for working with applications coupled to AMOS. The fast-path interface (FIF) and the embedded AMOSQL (EQL) can be used to communicate with the DBMS by applications with an embedded DBMS. AMOS also includes a foreign data source (FDS) interface for integrating foreign data and operator representations.*

It is of vital importance for the application to preserve the execution efficiency while adding functionality to the system. There are several factors that influence the overall performance in database systems where conventional engineering applications are combined with DBMSs. In the present approach execution efficiency is supported by main-memory processing, query optimization, and extensibility. Most important is the ability to have an embedded main-memory database where the application can access and update data through a fast-path interface using precompiled and preoptimized database functions.

Generally, execution efficiency is also supported by the query processor that has the ability to optimize access paths and operator ordering. This is especially important in complex modelling situations where the optimizer automatically can choose a good execution order. This simplifies the design of the database and frees the programmer from specifying the exact execution order which can be stated in higher-level terms. By providing the application with general and efficient data representations in the DBMS, these become directly available to the application and need not be re-implemented.

The AMOS extensibility with foreign data sources, i.e. packages of specialised data representations and operations, makes it possible to provide efficiency for critical activities. For instance, FEA involves large amounts of numerical data that must be represented and processed effectively. This requires data representations that are tuned for numerical operations, such as compact matrix representations.

Furthermore, specialised representations and operations in the DBMS are also required to avoid unnecessary duplication and transformation of data. If suitable data representations and operations are not available, data must be moved, and maybe transformed, to and from the DBMS for processing. Hence, the location of data and processing and data representation can be critical for processing efficiency, as also indicated in Stonebraker and Moore [6].

Three basic situations can be identified that reveals the problems of data and processing location:

1. Both data and operations are located in the application. This is one of the extreme cases where no DBMS is engaged and, accordingly, there are no DBMS capabilities and the location problem has no relevance.
2. Data is located in the database but the operations are located in the application. In a conventional R DBMS, where data is stored in tables, data might need both to be transformed to a suitable format and duplicated into the application. By using an extensible DBMS, the data transformation can be avoided but data must still be duplicated into the application for processing. The same holds for the opposite case and direction, but this case is probably not so common, where data is located in the application and the operations are available in the DBMS.
3. Both data and operations are located in the database. If appropriate data representations are available, data duplication and transformation can be avoided.

It should be noted, that the DBMS can provide indexing techniques that speed up data access. In general, it is also wise to make data reductions, such as filtering, within the database before data is transferred to the application. Hence, to take full advantage of the DBMS capabilities might require that certain parts of the application-specific operations should be performed by the DBMS. If this were solved for existing applications, it might imply a severe redesign of the application logic.

The size and number of data items must also be considered in evaluating where to locate data and operations. These kind of decisions, of where the processing should take place, could also be supported by the query processor. By defining appropriate cost measures for operators at different locations, the execution can directed to the one (or even several) of the servers in a network that is most cost-effective.

In FEAMOS, a manipulation of an application object through the TRINITAS user interface, or by an application procedure, implies an immediate update of the database. For example, when a point is moved on the screen it is a database point object that is updated. Thus, since application data is only stored in one place, data redundancy and inconsistency can be avoided.

With the availability of the query language it is further possible to query the contents in the database through the query language interface of the database. For example, we are currently using a www-interface in AMOS to connect a www-browser to the DBMS for interactive query access.

5.2 EXTENDING AMOS WITH LINEAR MATRIX ALGEBRA

One main idea in this work was to provide analysis capabilities in a database environment that can support the needs of scientific and engineering applications. As the usage of database techniques will increase in scientific and engineering disciplines the requirements on analysis capabilities will grow. Scientific and engineering applications are computationally intensive applications and the idea is to provide numerical analysis capabilities within the database environment to support the processing needs of these applications. The foreign data source for arrays, described in the next section, was designed and implemented with the intention to use it for implementing support for numerical linear algebra in AMOS.

By providing both data structures suitable for these applications and corresponding operations that are relevant it is possible to choose where the data should be located and where the processing should take place. Data transportation between the application and the database that are not required can be eliminated. Further, the types of analysis capabilities, together with the proposed architecture described in Section 4.2 and Section 5.1, that involve local embedded databases within applications, can provide new powerful techniques for developing scientific and engineering applications.

Hence, we would like to have numerical matrices in the database, not only the array data structure. This makes it possible to perform operations on matrices, producing new or modified matrices, compared to only accessing the physical array data structure. There are several research examples of how to provide this functionality in applications using a conventional approach, i.e. numerical analysis packages in programming languages. This includes both traditional programming languages, such as FORTRAN and C, Anderson et al. [129], as well as OO languages, such as C++, CommonLisp (including CLOS), OO dialects of Pascal, and Smalltalk, Baugh and Rehak [38], Fenves [39], Forde et al. [40], Filho and Devloo [41], Dubois-Pelerin et al. [42], Williams et al. [43], Scholz [44], Baugh and Rehak [45], Mackie [46], Ross et al. [47], Raphael and Krishnamoorthy [48], Yu and Adeli [49], Hoffmeister et al. [50], Arruda et al. [51], Devloo [52], Eyheramendy and Zimmermann [53], Gajewski [54], Ju and Hosain [55], Shepherd and Lefas [56], Langtangen [57], Cardona et al. [58], Zeglinski et al. [59], Lu et al. [60], Dongarra et al. [130], Dongarra et al. [131], and Barton and Nackman [132]. Our approach shares several ideas with the programming-language approach and further extends the ambitions in terms of expressibility and functionality by providing query-language integration and database facilities. Likewise, we extend the conventional database approach of only storing and accessing data with capabilities for numerical analysis within the DBMS. By having matrix types in the database it is possible to extend the query language with operations on matrices to form an algebra for the matrix domain that can be used in application modelling. It would further be possible to introduce special query optimization methods for numerical data processing to support automatic optimization of execution plans for numerical methods. This could, for instance, include decisions for selecting a suitable data representation, solution method, and processing location.

The need to include data storage types for collections of numerical data in DBMSs has been emphasised by several authors including Sarawagi and Stonebraker [133], Maier and Vance [3], Maier and Hanson [134], Vandenberg and DeWitt [135], Libkin et al. [136], Rotem and Zhao [137], and Seamons et al. [138]. In these works they mainly address the issue of providing numerical storage types from a secondary-storage perspective. An issue for future work would be to investigate how these techniques could be combined with the approach in this work focusing on main-memory representations.

5.2.1 Linear algebra for finite element analysis

A central problem in the present application domain of FEA is the solution of linear equation systems. This is a characteristic that is shared among many formulations of scientific and engineering problems. The solution process of linear equations is illustrated by an example from the FEA domain. First, the multiplication of general matrices can be expressed by the equational relation,

$$\mathbf{A}\,\mathbf{B} = \mathbf{C} \tag{50}$$

where matrix **A** times matrix **B** is equal to matrix **C**. If **A** and **B** are known and fulfil certain properties we can calculate **C** by matrix multiplication. In some cases it is also possible to calculate **A** or **B** if the two other matrices are known. However, this is not as straightforward as in the former case since it usually involves the solution of some kind of equation system.

A specific case, but central in scientific and engineering computations, is the case where Eq. (50) forms a linear equation system. In this case, the equation system is composed of one square matrix **A**, and two column matrices **b** and **c**.

In the FEA domain, the solution of these types of equation systems is a central issue. A common problem in this domain is to solve an equation (stated earlier in Section 2.3 as Eq. (30)) of the form

$$\mathbf{K}\,\mathbf{a} = \mathbf{f}\,. \tag{51}$$

Recalling from Section 2.3, that in the treatment of linear elasticity problems, **K** is usually called the stiffness matrix, **a** the displacement vector, and **f** the load vector. Further, in linear elasticity, the stiffness matrix is symmetric and non-singular. If appropriate boundary conditions are enforced and the equations corresponding to fixed displacements are eliminated, the resulting submatrix of **K** will become positive definite.

These types of linear systems are usually solved by transforming the equation system into one or several equation systems that are simpler, but equivalent, to solve. In the FEA domain, this process usually involve a decomposition of **K** into a set of matrices. There are various decomposition methods, but a common method in FEA decomposes **K** into three matrices as

$$\mathbf{K} = \mathbf{L}\,\mathbf{D}\,\mathbf{U} \tag{52}$$

where **L** is a lower unit triangular matrix, **D** is a diagonal matrix, and **U** is an upper unit triangular matrix. Unit triangular matrices have unit values in its diagonal. Furthermore, for non-singular symmetric matrices it holds that $\mathbf{U} = \mathbf{L}^{T}$, Golub and van Loan [139]. This decomposition method is called LDL^{T} decomposition, or Crout decomposition. Further, due to the symmetry, this decomposition reduces the amount of computation work to perform. It can also be computed without engaging the right-hand side of Eq. (51) in contrast to conventional Gauss elimination techniques. Alternative decomposition methods can be added to cover additional problem classes.

Hence, Eq. (51) can be transformed into the equivalent form

$$\mathbf{U}^{T}\,\mathbf{D}\,\mathbf{U}\,\mathbf{a} = \mathbf{f}\,. \tag{53}$$

Now, by introducing the intermediate quantities **y** and **z**, Eq. (53) can be solved by,

$$\begin{aligned} \mathbf{U}\,\mathbf{a} &= \mathbf{y} \\ \mathbf{D}\,\mathbf{y} &= \mathbf{z} \\ \mathbf{U}^T\,\mathbf{z} &= \mathbf{f} \end{aligned} \qquad (54)$$

where the first and last equation systems are triangular systems and the mid system is a diagonal system or linear scaling. As we shall see later, Eqs. (54) can be solved in a straightforward manner.

5.2.2 Matrix algebraic concepts

Some matrix algebraic concepts will be introduced where the notation mainly follows that of Golub and van Loan [139]. The vector space of all m-by-n matrices is denoted by the m-by-n scalar field $S^{m \times n}$, where normally $S \in R$, the set of real numbers. However, due to the computational requirements it might be necessary to extend the types of matrix representations such that S belongs to one of Z (the set of integers), R_f (the set of four-byte reals), and R_d (the set of 8-byte reals). This means that matrices can have integer, float, and double representations. If matrix expressions were to allow mixing matrix types, this must be taken care of in the definition of matrix operations. This distinction is left out in the subsequent presentation of matrix concepts.

Thus, for a matrix **A** we have,

$$\mathbf{A} \in S^{m \times n} \Leftrightarrow \mathbf{A} = (a_{ij}) = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}, \text{ where } a_{ij} \in S \qquad (55)$$

Here, a_{ij} represents the element of **A** at row i and column j.

Basic matrix algebraic operations of matrices can now be introduced. The conventional approach introduces matrix algebraic operations as functions that take matrices as arguments and produce new or altered matrices. Here, a somewhat different approach will be applied. Since the query language AMOSQL allows the definition of multi-directional functions, it is possible to define operations on matrices as relationships that are isomorphic to the corresponding mathematical expressions. In Golub and van Loan, [139], operations are represented by the $a \to b$ notation, where the arrow associates to a one-directional function application. In the present context, this notation is replaced by the $a \leftrightarrow b$ notation that is more associated to bi-directional or multi-directional relationships. However, it should be noted that this notation does not imply that the relationship exist, or is defined, for every direction that corresponds to the combinations of matrix types.

Hence, the basic operations on matrices include:

- **addition**:
  $S^{m \times n} \times S^{m \times n} \leftrightarrow S^{m \times n}$, where $\mathbf{A} + \mathbf{B} = \mathbf{C}$ with the elements $a_{ij} + b_{ij} = c_{ij}$
- **subtraction**:
  $S^{m \times n} \times S^{m \times n} \leftrightarrow S^{m \times n}$, where $\mathbf{A} - \mathbf{B} = \mathbf{C}$ with the elements $a_{ij} - b_{ij} = c_{ij}$
- **multiplication**:
  $S^{m \times r} \times S^{r \times n} \leftrightarrow S^{m \times n}$, where $\mathbf{A} \cdot \mathbf{B} = \mathbf{C}$ with the elements $\sum_{k=1}^{r} a_{ik} \cdot b_{kj} = c_{ij}$
- **transposition**:
  $S^{m \times n} \leftrightarrow S^{n \times m}$, where $\mathbf{A}^T = \mathbf{B}$ with the elements $a_{ji} = b_{ij}$

There are also operations on combinations of scalars and matrices that include:

- **multiplication** of scalar and matrix:
  $S \times S^{m \times n} \leftrightarrow S^{m \times n}$, where $a \cdot \mathbf{B} = \mathbf{C}$ with the elements $a \cdot b_{ij} = c_{ij}$
- **multiplication** of matrix and scalar:
  $S^{m \times n} \times S \leftrightarrow S^{m \times n}$, where $\mathbf{A} \cdot b = \mathbf{C}$ with the elements $a_{ij} \cdot b = c_{ij}$
- **division** of scalar and matrix:
  $S^{m \times n} \times S \leftrightarrow S^{m \times n}$, where $\mathbf{A}/b = \mathbf{C}$ with the elements $a_{ij}/b = c_{ij}$

In addition we have the normal scalar arithmetic operations:

- **addition**: $S \times S \leftrightarrow S$, where $a + b = c$
- **subtraction**: $S \times S \leftrightarrow S$, where $a - b = c$
- **multiplication**: $S \times S \leftrightarrow S$, where $a \cdot b = c$
- **division** of scalar and matrix: $S \times S \leftrightarrow S$, where $a/b = c$

We should note that the matrix concept defined above covers general m-by-n matrices. By making restrictions on this definition it is possible to define specialised categories of matrices that forms subspaces of the vector space $S^{m \times n}$. For instance, we can define

- $S^{m \times n}$, representing the general **rectangular** matrix, $\mathbf{A}_{rect}$.

- $S^{m \times m}$, representing a **square** matrix, $\mathbf{A}_{square}$, with the same number of rows and columns.

- $S^{m \times m}$, a square matrix with the additional constraint $s_{ij} = s_{ji}$ that represents a **symmetric** matrix, $\mathbf{A}_{symm}$.

- $S^{m \times m}$, a symmetric matrix with the additional constraint $s_{ij} = 0$ for $i \neq j$ that represents a **diagonal** matrix, $\mathbf{A}_{diag}$.

- $S^{m \times m}$, a matrix with the same number of rows and columns with the additional constraint $s_{ij} = 0$ for $i > j$ and that represents an **upper triangular** matrix, $\mathbf{A}_{uptri}$.

- $S^{m \times m}$, an upper triangular matrix with the additional constraint $s_{ij} = 1$ for $i = j$ that represents an **upper unit triangular** matrix, $\mathbf{A}_{uputri}$.

- $S^{m \times m}$, a matrix with the same number of rows and columns with the additional constraint $s_{ij} = 0$ for $i < j$ and that represents a **lower triangular** matrix, $\mathbf{A}_{lowtri}$.

- $S^{m \times m}$, an upper triangular matrix with the additional constraint $s_{ij} = 1$ for $i = j$ that represents a **lower unit triangular** matrix, $\mathbf{A}_{lowutri}$.

- $S^{m \times 1}$, a rectangular matrix with 1 column representing a **column** matrix, $\mathbf{a}$ or $\mathbf{A}_{col}$.

- $S^{1 \times m}$, a rectangular matrix with 1 row representing a **row** matrix type, $\mathbf{a}$ or $\mathbf{A}_{row}$.

With these additional categories of matrices, the previous list of matrix operations can also be specialised further taking the additional categories into account. This is illustrated here for the case of matrix multiplication of rectangular matrices. By identifying index sizes of the arguments and the results the following combinations arise:

- $S^{m \times r} \times S^{r \times n} \leftrightarrow S^{m \times n}$, or matrices categorised as $\mathbf{A}_{rect} \cdot \mathbf{B}_{rect} = \mathbf{C}_{rect}$

- $S^{m \times n} \times S^{n \times n} \leftrightarrow S^{m \times n}$, or matrices categorised as $\mathbf{A}_{rect} \cdot \mathbf{B}_{square} = \mathbf{C}_{rect}$

- $S^{n \times n} \times S^{n \times m} \leftrightarrow S^{n \times m}$, or matrices categorised as $\mathbf{A}_{square} \cdot \mathbf{B}_{rect} = \mathbf{C}_{rect}$
- $S^{m \times 1} \times S^{1 \times n} \leftrightarrow S^{m \times n}$, or matrices categorised as $\mathbf{A}_{col} \cdot \mathbf{B}_{row} = \mathbf{C}_{rect}$
- $S^{m \times n} \times S^{n \times m} \leftrightarrow S^{m \times m}$, or matrices categorised as $\mathbf{A}_{rect} \cdot \mathbf{B}_{rect} = \mathbf{C}_{square}$
- $S^{m \times m} \times S^{m \times m} \leftrightarrow S^{m \times m}$, or matrices categorised as $\mathbf{A}_{square} \cdot \mathbf{B}_{square} = \mathbf{C}_{square}$
- $S^{m \times 1} \times S^{1 \times m} \leftrightarrow S^{m \times m}$, or matrices categorised as $\mathbf{A}_{col} \cdot \mathbf{B}_{row} = \mathbf{C}_{square}$
- $S^{m \times n} \times S^{n \times 1} \leftrightarrow S^{m \times 1}$, or matrices categorised as $\mathbf{A}_{rect} \cdot \mathbf{B}_{col} = \mathbf{C}_{col}$
- $S^{n \times n} \times S^{n \times 1} \leftrightarrow S^{n \times 1}$, or matrices categorised as $\mathbf{A}_{square} \cdot \mathbf{B}_{col} = \mathbf{C}_{col}$
- $S^{1 \times n} \times S^{n \times m} \leftrightarrow S^{1 \times m}$, or matrices categorised as $\mathbf{A}_{row} \cdot \mathbf{B}_{rect} = \mathbf{C}_{row}$
- $S^{1 \times n} \times S^{n \times n} \leftrightarrow S^{1 \times n}$, or matrices categorised as $\mathbf{A}_{row} \cdot \mathbf{B}_{square} = \mathbf{C}_{row}$
- $S^{1 \times n} \times S^{n \times 1} \leftrightarrow S^{1 \times 1}$, or matrices categorised as $\mathbf{A}_{row} \cdot \mathbf{B}_{col} = c$
- $S^{1 \times 1} \times S^{1 \times 1} \leftrightarrow S^{1 \times 1}$, or matrices categorised as $a \cdot b = c$

Hence, the resulting matrix category of multiplying two (rectangular) matrices is dependent on the sizes of the outer indexes of the argument matrices. By interpreting the above matrix spaces as sub-categories of the rectangular matrix category we get relationships between argument matrix categories and result argument category for the matrix multiplication operator. By considering other matrix characteristics, such as symmetry and singularity, further specialisations of these relationships can be established. For instance, if it is known that a certain matrix is symmetric and it is multiplied with another matrix, the appropriate result category can be produced. However, if we only know that the matrix is square, the symmetry condition must be evaluated before this information can be used in deciding the result category.

Furthermore, in applications like FEA, it is common to use specialised and more compact physical representations, Hughes [23], and Carey and Oden [140], of matrices in contrast to “full” *regular matrix* representations. As illustrated in Figure 20, there are, for example, *skyline matrix* (or profile matrix) representations where consecutive zero-valued elements above the “skyline” are left out and the matrix is usually represented by matrix columns in a one-dimensional array. This is an example of a compact representation where the matrix structure is static, i.e. it is not allowed to change. Additional static representations along the same theme exists. There are also dynamic matrix representations where the storage structure is allowed to change. These representation types are usually referred to as *sparse matrix* representations and are typically implemented by some linked-list data structure. The categorisations and their usage in establishing the multiplication operator as relationships among different categories that are exemplified above can be further extended to establish relationships between combinations of other matrix categories and representations as well as for different operators.

To sum up this part, three principles have been presented that divide the matrix concept into different categories, namely:

- mathematically-related matrix categories based on the general matrix concept and its characteristics that further restrict this concept into subcategories.
- the data types, integer, float, and double, used for representing and implementing numerical matrices.
- various physical representations schemes for representing and implementing matrices such as regular, skyline, or sparse. The present representations are restricted to the regular and skyline type.

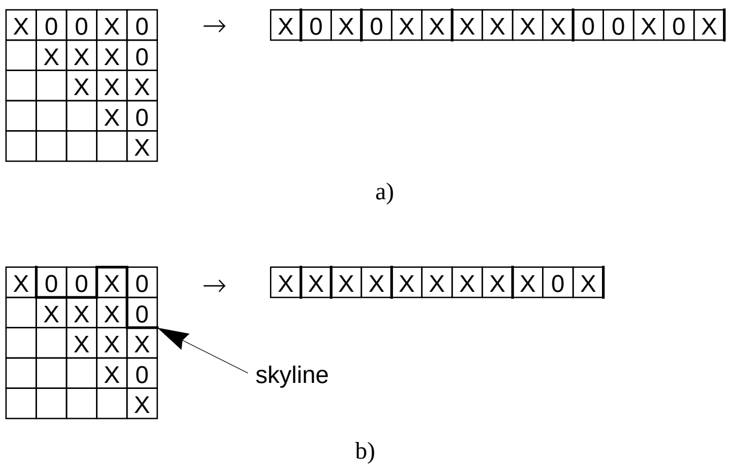


Figure 20. *Examples of two different one-dimensional physical representations for symmetric and triangular matrices, where X indicates non-zero values. In a), a full storage scheme is illustrated where each element is stored. In the skyline storage scheme, illustrated in b), zeros over the skyline are excluded.*

The reason for defining several matrix categories is the potential ability to take advantage of the knowledge about specific categories in representing numerical data and applying numerical analysis methods. This concerns the possibilities of applying efficient storage and processing techniques. So far it is possible to use:

- a priori information to determine matrix categories appropriate for a specific problem.
- information about matrix categories related by a specific operator to determine appropriate operator and the correct result (or argument) category.
- information about matrix characteristics, i.e. properties that are not distinguished by separate categories, to determine correct operator result (or argument) to efficiently direct subsequent matrix representations and operations.

The next section shows how these matrix concepts can be represented in, and managed by, the AMOSQL extensible and OO query language.

If declarative matrix algebraic expressions do not contain a unique execution order, it is possible to apply some form of operator reordering to obtain an efficient processing. This is analogous to applying query optimization techniques where initial query expressions are transformed into equivalent but more efficient execution plans. Optimization of the execution order for matrix algebraic expressions must be based on the mathematical laws of matrix algebra that restrict the way that these expressions can be transformed. In the general case, it should be possible to extend the query optimization techniques with capabilities to optimize general query expressions that include matrix algebraic operators. Similar ideas have been suggested by Wolniewicz and Graefe in [142] for integration of mathematical operations on time series.

For instance, it would be possible to apply the laws for scalar-matrix and matrix-matrix multiplication, [143], to guide the execution order.

$c(\mathbf{A}\,\mathbf{B}) = (c\mathbf{A})\mathbf{B} = \mathbf{A}(c\mathbf{B})$	associativity of scalar and matrix multiplication
$\mathbf{A}(\mathbf{B}+\mathbf{C}) = \mathbf{AB}+\mathbf{AC}$	distributive law of premultiplication
$(\mathbf{B}+\mathbf{C})\mathbf{D} = \mathbf{BD}+\mathbf{CD}$	distributive law of postmultiplication
$(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$	associative law of matrix multiplication

This can be exemplified by the last law, i.e. the associative law of matrix multiplication, that can be expressed in an operator tree as in Figure 21. By using knowledge about types and further matrix properties, a preferred execution order can be decided upon.

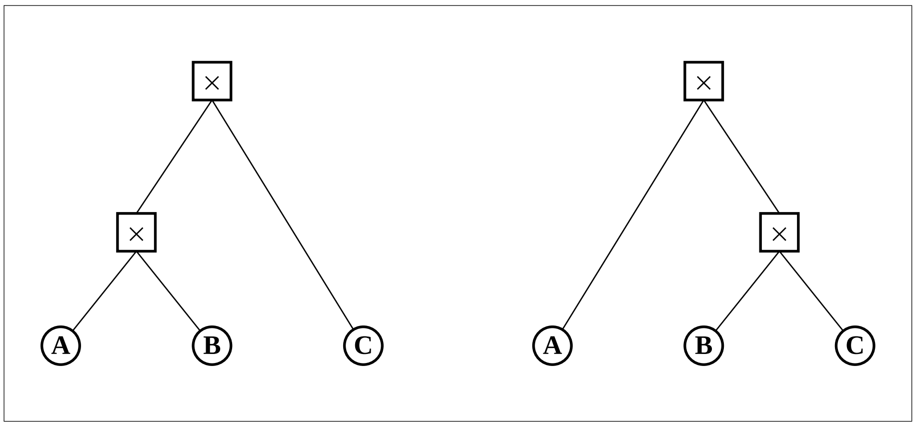


Figure 21. *Operator trees for the associative law of matrix multiplication where the two trees represent corresponding operations.*

The preferred operator order can be determined by comparing the estimated costs corresponding to operator sequences. This is exemplified by considering the multiplication of three rectangular matrices. The problem is to find the most efficient execution order of the following combination of matrix multiplications,

$$\mathbf{A} \cdot \mathbf{B} \cdot \mathbf{C} \text{, where } \mathbf{A} \in S^{m \times n}, \mathbf{B} \in S^{n \times k}, \mathbf{C} \in S^{k \times l} . \tag{56}$$

with the two possible operator sequences that were illustrated in Figure 21. Thus we can either apply the execution order,

$$(\mathbf{AB})\mathbf{C} \text{ , or} \tag{57}$$

$$\mathbf{A}(\mathbf{BC}) . \tag{58}$$

The estimated costs for matrix multiplication is proportional to the number of multiplications of scalar values that are involved. For a simple matrix multiplication of two rectangular and regular matrices the cost, C^M, can be expressed as:

$$C^M = C^M(m, n, k) = O(mnk) \sim C_0^M mnk \tag{59}$$

where M represents multiplication, *m* and *n* is the size of the first matrix and *n* and *k* of the second. Further, C_0^M, represents a general constant related to the matrix multiplication operator. For combinations of several multiplications we get the total cost, C^M_{tot}, from:

$$C^M_{tot} = \sum_i C^M_i(m_i, n_i, k_i) \tag{60}$$

where the summation is made over *i*, i.e. all simple multiplication operations. The preferred order can then be found by determining the one with the minimum total cost. Continuing our example, if we consider C^M_0 as constant over each simple operation, we can calculate the costs for the two possible execution orders as:

$$C^{M_I}_{tot} = C_0(mnk + mkl) \tag{61}$$

$$C^{M_{II}}_{tot} = C_0(mnl + nkl) \tag{62}$$

where (I) is the case given by (57) and (II) the case in (58). If, for example, we choose $m = k = 1$ and $n = l = 10$, we get $C^{M_I}_{tot} = 20C_0$ and $C^{M_{II}}_{tot} = 200C_0$. This example is illustrated in case a) in Figure 22 and tells us that the first execution order, given by (58), is preferable. On the other hand, if we choose $m = k = 10$ and $n = l = 1$ we get $C^{M_I}_{tot} = 200C_0$ and $C^{M_{II}}_{tot} = 20C_0$; pointing out the first execution order, given by (57), as favourable. This corresponds to case b) in Figure 22.

Instead of calculating actual costs to determine the preferable execution order, one can use the difference in execution cost, ΔC^M_{tot}, as the starting point. Hence, we have

$$\Delta C^M_{tot} = C^{M_{II}}_{tot} - C^{M_I}_{tot} = C_0(mn(l-k) + kl(n-m)) \quad . \tag{63}$$

Since each quantity in Eq. (63) is non-negative, it is possible to state that for $\mathbf{A} \cdot \mathbf{B} \cdot \mathbf{C}$ where $\mathbf{A} \in S^{m \times n}$, $\mathbf{B} \in S^{n \times k}$, $\mathbf{C} \in S^{k \times l}$ and,

i) if $(l-k) > 0 \wedge (n-m) > 0$ choose execution order $(\mathbf{AB})\mathbf{C}$ (illustrated by case a) in Figure 22), or

ii) if $(l-k) < 0 \wedge (n-m) < 0$ choose execution order $\mathbf{A}(\mathbf{BC})$ (illustrated by case b) in Figure 22), or

iii) if $(l-k) = 0 \wedge (n-m) = 0$ the decision is indifferent (exemplified by case c) and d) in Figure 22, or

iv) if $(l-k) < 0 \wedge (n-m) > 0$ or $(l-k) > 0 \wedge (n-m) < 0$ the decision is indefinite and further examinations of the costs must be performed by evaluating the total costs measure in (63). This resulting decision can be either of i, ii, or iii. Examples of this situation are presented in case e) and f) in Figure 22, of which the conclusion in these special cases is that they are indifferent.

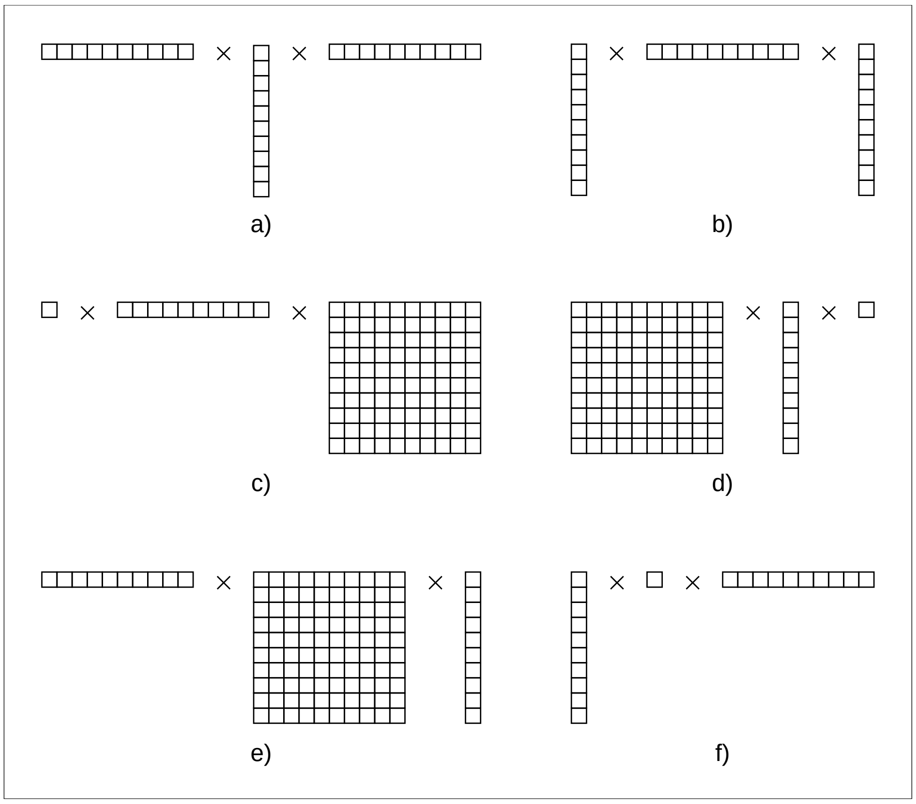


Figure 22. *Examples of how the execution order and matrix structure affects the cost for consecutive matrix multiplications.*

In general, it is preferable to execute contracting operators before expanding operators, but for combined operations the total cost must be considered. Similar preference orderings can be made for the other multiplication laws and additional operators and operator combinations. Furthermore, there are other factors that can affect the execution cost and can be introduced into a cost model, such as the representation scheme and memory requirements. This last type of optimization technique suggested, cost-based optimization of matrix operators, is not yet implemented in the FEAMOS system.

Query optimization can be made both at compile time and at run time. Predefined queries can be optimized at compile time if the necessary information is available whereas ad hoc queries must be optimized at run time. Since query optimization normally is an expensive operation, it is preferably to perform it at compile time if possible. For example, the join operation is a commutative operation that results in an exponential increase in possible query plans when several join operations are combined.

It is further possible to use information available at compile time, e.g. type information or other problem-specific knowledge (such as that the linear-elastic stiffness matrix is symmetric and non-singular), to guide optimization of matrix expressions at compile time. The type system can be used to select the correct execution plan as long as it is unique, and this is the technique currently being used for solving linear equations.

Since data is not static, conventional query optimization normally uses statistics about data, such as table sizes and predicate selectivities for performing optimization at compile time. In analogy with this type of query optimization, it would be possible in the matrix domain to store statistics about average matrix sizes and later apply this knowledge in query optimization and compilation to select efficient execution methods. The applicability of this technique must be further investigated.

To be able to explore actual sizes and structures of matrices, properties that vary from one analysis case to another, the optimization must take place at run time. If there are a few possible execution plans or other simple decisions to be made this might be satisfactory with respect to efficiency. For instance, the associative law of matrix multiplication, considered in the former example, represents a lower combinatorial complexity than the commutative relational join operation that has factorial complexity. For factorial complexity we have:

$a_n = n!$, where $n! \sim \sqrt{n}\left(\frac{n}{e}\right)^n$, according to Stirling’s formula [144].

In the case of the associative law of matrix multiplication we have:

$$a_n = \left(1 - \frac{3}{2\sqrt{2}}\right)(1 - \sqrt{2})^{n-2} + \left(1 + \frac{3}{2\sqrt{2}}\right)(1 + \sqrt{2})^{n-2} \quad .$$

Hence, matrix multiplication exhibits a lower exponentiality than the join operation. This can indicate that matrix expressions of reasonable size that include a small number of alternative execution plans, or simple decisions, could be optimized at run-time. Before any further conclusions can be made, additional investigations must be made in this matter.

However, if run-time optimization is not satisfactory, *dynamic optimization* can be applied, Cole and Graefe [145], that handles precompiled alternative execution plans. By precompiling several alternative execution plans, a specific execution plan can be selected at run time at a much smaller cost than if pure run-time optimization was required.

5.2.3 The matrix foreign data source

If matrix algebraic operators are integrated within the query language it is possible to perform numerical analysis expressed as declarative queries. The query language then needs to be extended with:

- A set of basic matrix types and
- A set of basic matrix operations.

It is then possible to use the built-in capabilities of the query language to:

- Perform domain modelling and manipulation where domain concepts are based on matrix types. Hence, physical conversion between the application and the database, such as copying and transformation of data, can be avoided since the same physical formats can be supplied at both locations. Having domain concepts in the database representation that are isomorphic to application concepts also avoids unnecessary conceptual conversions between concepts in the application and in the database.
- Make queries that involve matrices in combination with other types of data, i.e. queries can be stated that combine and process matrix data with other data types (heterogeneous data in general). A uniform representation and access to matrix data will also facilitate the combination of matrix-based data across application boundaries.
- Express composite, and more complex, matrix operations in terms of the basic set of matrix operations. Further, algorithms can be expressed in a data independent manner in terms of these basic or composite operations. For instance, multi-directional operations can provide reuse of operations, which results in concise algorithm representations.
- Declarative expressiveness can leave execution order decisions to the query processor. Matrix types can be used to control processing of matrix operations. Matrix operations can be made polymorphic such that the choice of a specialised and appropriate algorithm is dependent on the problem type. By having specialised storage schemes and operators for different matrix types it is possible to guide the numerical computation into efficient processing alternatives and to choose optimized storage and processing techniques where possible.
- The query processor can be extended to understand domain-specific operators (queries) that provide algorithm-choices based on cost measures. Hence, the application domain can take advantage of algebraic and cost-based optimization of matrix, or other, domain operations.

Furthermore, one would like to have appropriate database representations of matrices with respect to storage and processing concerns. Tailored storage schemes and algorithms to optimize storage and processing requirements are common in FEA. For these reasons, it would be convenient to exploit matrix characteristics based on mathematical properties including structure, physical matrix representation schemes, and basic data types.

To accomplish the above functionality, it is required (or at least most facilitating) that the query language is equipped with certain capabilities. These include:

- Object-oriented features such as user types, subtyping, inheritance, etc., as stated in Section 3.3.1. Object-orientation provides facilities for structuring the matrix domain in a problem-oriented fashion while reusing specifications in the type taxonomy. The type system also provides the capability to optimize processing by selecting algorithms based on type information.
- Overloading is required to be able to define variants of matrix operations specialised for matrix sub-categories or matrix representations.
- Overloading on all arguments makes it possible to have functions specialised for combinations of matrix types that is typically the case for matrix operations.
- Multi-directional functions provide the ability to apply functions in multiple directions, facilitating reuse of functions and compact representations.
- Foreign functions are a basic capability for transparently integrating matrix operations within the query language while implementing numerical matrix operations efficiently in a programming language.
- Customized internal data representations are, together with foreign functions, required to be able to handle the representation and processing of specialised matrix representation schemes.
- To be able to direct decisions concerning execution order, selection of redundant algorithms, and others, to the query processor, cost-based optimization must be supported.
- Main-memory DBMS technology provides efficient processing capabilities that are vital for numerical analysis computations as performed in FEA.
- Support of integration, combination, and exchange of data from other DBMSs.

In these issues, there are several problems that need to be solved. In pure object-orientation only the receiver of a message, corresponding to the first argument of a function, is used for deciding the appropriate method to apply. This is not sufficient to conveniently define mathematical operations, such as matrix algebraic operations, since operations need to be defined for different combinations of argument types in a type hierarchy. For example, we can define plus (+) and times (*) for different combinations of numbers and matrices as:

```
plus(number n1, number n2) -> number n3
plus(matrix m1, matrix m2) -> matrix m3
```

```
times(number n1, number n2) -> number n3
times(number n1, matrix m1) -> matrix m2
times(matrix m1, number n1) -> matrix m2
times(matrix m1, matrix m2) -> matrix m3
```

We see that for the `plus` function it is sufficient with overloading on the first argument to be able to distinguish between the two variants. In the other case, where the `times` function is overloaded, a selection mechanism based on the first argument is not sufficient since there are several possible functions to choose among.

However, there are several examples where matrix class libraries are defined in OO programming languages like C++, Scholz [44], Ross et al. [47], Yu and Adeli [49], Zeglinski et al. [59], Lu et al. [60], Dongarra et al. [131], and Barton and Nackman [132]. These types of languages[1] do not permit an automatic treatment of overloading of operations on multiple arguments and the operation dispatch must be handled explicitly. An alternative solution would be to have special operators for each case and by providing overloading on all arguments this can be achieved, Flodin et al. [19]. In comparison to pure object-orientation this technique extends the capabilities for abstraction and reuse.

Furthermore, it would be desirable to be able to apply matrix operations in different directions in queries and functions. A simple case is seen in the definition of a matrix transposition function in the example below

```
create function transpose(lower_triangular_matrix ltm) ->
                          upper_triangular_fmatrix utm as
                    select utm where transpose(utm) = ltm;
```

where transposition of a `lower_triangular_matrix` is accomplished by applying the inverse of transposing an `upper_triangular_fmatrix`.

The ability to handle multi-directional functions provides several advantageous features:

- multi-directional functions make it possible to reuse operations. An example of how this is applied for reusing matrix operations is given below. The minus operator, in the example, is expressed in terms of a formerly defined plus operator. In this example, the first argument of the plus function is unbound and the appropriate application of the multi-directional plus function will be handled by the system.

  ```
  create function minus(matrix m1, matrix m2) -> matrix m3 as
                    select a3 where plus(m3,m2) = m1;
  ```

- multi-directional applicability of functions can significantly reduce database search,

1. C++ can handle overloading on all arguments if late binding does not occur.

reported in Flodin [87]. Multi-directional functions and indexes increase the physical data independence since the system can efficiently execute a function in its most efficient direction independently from its logical usage.

- multi-directional functions can increase the readability and simplify the application design since the problem description becomes more domain-oriented and implementation independent.

The following example shows how the solution of a triangular equation systems can be expressed in a straightforward and mathematically-oriented manner. A triangular equation system is composed by multiplying a triangular matrix and a column matrix; forming a second column matrix. This can be expressed by the multiplication function, times, that is overloaded for different types of triangular matrices illustrated by

$$S^{n \times n} \times S^{n \times 1} \leftrightarrow S^{n \times 1}$$

where the first matrix represents a generic triangular matrix category that has several specialisations that correspond to the following multiplication operations:

$$\begin{aligned} \mathbf{A}_{uptri} \cdot \mathbf{B}_{col} &= \mathbf{C}_{col} \\ \mathbf{A}_{uputri} \cdot \mathbf{B}_{col} &= \mathbf{C}_{col} \\ \mathbf{A}_{lowtri} \cdot \mathbf{B}_{col} &= \mathbf{C}_{col} \\ \mathbf{A}_{lowutri} \cdot \mathbf{B}_{col} &= \mathbf{C}_{col} \end{aligned} \tag{64}$$

There are additional variants of these operations since the matrix categories can have different representation schemes and basic numerical data type. In AMOSQL, the multiplication operators are implemented as multi-directional foreign functions that are defined for appropriate binding patterns. A binding pattern is a unique pattern that states which of the interface variables of a function are presumed to be bound and which that are free (unbound). The current implementation of the times function include the following two binding patterns. The first is

$$\mathbf{A}^b \cdot \mathbf{B}^b = \mathbf{C}^f \tag{65}$$

where indexes represent if the function variable is bound (*b*) or free (*f*). Here, the two arguments are bound and the result is free. This binding pattern corresponds to normal multiplication. Further, operations that correspond to the second binding pattern,

$$\mathbf{A}^b \cdot \mathbf{B}^f = \mathbf{C}^b \tag{66}$$

are not always defined. However, when the second argument is a column matrix, it corresponds to the solution of different types of linear equation systems (including triangular systems). Here the first argument and the result are bound and the second argument is free, i.e. the unknown in the equation. The multiplication operators that are currently implemented are listed in Appendix C.

By defining overloaded foreign functions for different binding patterns, the number of operations that are required for expressing the same functionality can be reduced. The same operation can be reused in application design in a uniform and compact fashion. Furthermore, operations such as matrix algebraic operations can be expressed in a mathematically tractable syntax. For instance, all of the operations that correspond to Eqs. (64) for solving triangular equation systems form a subset of all the operations that can be expressed by the same AMOSQL expression as,

```
select b for each column_matrix b where :A * b = :c;
```

where `:A` and `:c` are variables bound to matrix OID:s. Defining and composing various matrix operations to form more complex expressions that will be executable by the query processor is also allowed. The following query includes a transpose function that will influence the decision of which multiplication operator to apply.

```
select b for each column_matrix b where transpose(:A) * b = :c;
```

If `:A` is bound to an upper triangular matrix, the transpose operator will change the type of `:A` to a lower triangular matrix and another multiplication operator will be applied than if the transpose operation were absent.

As an example of how overloaded multiplication functions can be defined and used we consider the first and third equation in Eqs. (64). The first equation can be interpreted as corresponding to a matrix multiplication when **A** and **B** are known, and to solving an upper triangular linear equation system when **A** and **C** are known. For the situation where **B** and **C** are known the equation has no unique solution. This interpretation can be captured by overloading a function for matrix multiplication of an upper triangular matrix with a column matrix that results in another column matrix. This function should be overloaded on the *bbf* binding pattern and for the *bfb* binding pattern. By leaving out the *fbb* binding pattern, the result will be undefined for this case. This can be defined in AMOSQL by the following two functions.

```
create function times(upper_triangular_matrix uptri bound,
                      column_matrix cola bound) ->
                      column_matrix colr free as
                           foreign times.uptri.col.col--+fn;

create function times(upper_triangular_matrix uptri bound,
                      column_matrix cola free) ->
                      column_matrix colr bound as
                           foreign times.uptri.col.col-+-fn;
```

In this case, the actual behaviour is implemented by foreign functions in Lisp and C.

Similarly, the third equation in Eqs. (64) corresponds to a matrix multiplication when **A** and **B** are known, the solution of a lower triangular equation system when **A** and **C** are known, and when **B** and **C** there is no unique solution. This is translated to the two AMOSQL functions:

```
create function times(lower_triangular_matrix lowtri bound,
                      column_matrix cola bound) ->
                      column_matrix colr free as
                           foreign times.lowtri.col.col--+fn;

create function times(lower_triangular_matrix lowtri bound,
                      column_matrix cola free) ->
                      column_matrix colr bound as
                           foreign times.lowtri.col.col-+-fn;
```

These functions can be used in defining additional matrix functions in AMOSQL. For instance, it might be convenient to define a function for solving triangular systems that can be precompiled and called directly from an application. This is accomplished by the lin_solve AMOSQL function:

```
create function lin_solve(triangular_matrix tri,
                          column_matrix cola) -> column_matrix colr as
                    select colr where tri * colr = cola;
```

By additional multiplication function definitions the `lin_solve` function will apply to all triangular systems in Eqs. (64). As we shall see later, it is possible to generalise this concept even further.

Hence, it has been showed how matrix operations can be defined and applied using overloaded multi-directional functions and foreign functions. We now turn to another example to show how AMOSQL can be further extended with additional matrix operations to solve linear equation system. Referring to Eq. (30) in Section 2.3, a linear equation system of a typical finite element analysis, such as in static linear-elastic stress analysis, can be expressed as,

$$\mathbf{K}^b \cdot \mathbf{a}^f = \mathbf{f}^b \tag{67}$$

where **K** is a symmetric and non-singular matrix, **a** is a column vector of unknowns, and **f** is a column vector. This can be expressed by a multiplication function with the first argument and the result bound and the second argument free. Thus, solving the equation system Eq. (67) can be very easily expressed in AMOSQL as,

```
select a for each column_matrix a where :k * a = :f;
```

where the resulting column vector cx is calculated by applying an appropriate form of the multi-directional multiplication function which is determined by the query processor. Looking at this in more detail, the definition the multiplication function for this binding pattern involves the application of three additional and simpler multiplication operators and a factorisation operator that forms the complete solution of a linear equation system. Again referring to Eqs. (52), (53), and (54) in this section, this solution process can be declaratively expressed by functions corresponding to the equational expressions,

$$\begin{aligned} \mathbf{K}^b &= (\mathbf{U}^T)^f \cdot \mathbf{D}^f \cdot \mathbf{U}^f \\ \mathbf{U}^b \cdot \mathbf{a}^f &= \mathbf{x}^b \\ \mathbf{D}^b \cdot \mathbf{x}^f &= \mathbf{y}^b \\ (\mathbf{U}^T)^b \cdot \mathbf{y}^f &= \mathbf{f}^b \end{aligned} \qquad (68)$$

where **U** is an upper unit triangular matrix, **D** is a diagonal matrix, and **x** and **y** are column vectors. When **K** and **f** are known, the execution order is implicitly given from the binding patterns. Hence, by defining the operation corresponding to Eq. (67) as the solution of Eq. (68), the solution of the a linear equation system can be expressed as the inverse to a single matrix multiplication expression. The query processor will then automatically transform expressions of the form in Eq. (67) to a corresponding representation in Eq. (68). This means that the function for multiplying a symmetric matrix with a column matrix has a straightforward implementation for the *bbf* binding pattern as,

```
create function times(symmetric_matrix k bound,
                      column_matrix a bound) ->
                      column_matrix f free as
                          foreign times.sym.col.col--+fn;
```

where `times.sym.col.col--+fn` is the foreign function that should be executed. For the *bfb* binding pattern we should express the transformation of Eq. (67) to Eq. (68) which is accomplished by the following function definition:

```
create function times(symmetric_matrix k bound,
                      column_matrix a free) ->
                      column_matrix f bound as
                   select a
                      for each
                          upper_unit_triangular_matrix u,
                          diagonal_matrix d,
                          column_matrix ca, column_matrix cf
                      where
                          factorise(k) = <u,d> and
                          transpose(u) * cf = f and
                          d * ca = cf and
                          u * a = ca;
```

As mentioned, there is no explicit execution order in function definitions; it is determined by the query processor. For instance, in the solve function above, this means that the subexpressions in the where clause can be arbitrarily ordered.

If one would like to access the analysis capabilities of the DBMS from an application, the AMOSQL expression for solving the equation can be packed into a precompiled function that can be directly called from the application. This would look like

```
create function lin_solve(symmetric_matrix k,column_matrix f) ->
                        column_matrix a as
                     select a where k * a = f;
```

and which can be called by,

```
lin_solve(:k,:f);
```

and where `:k` and `:f` are references to matrix OID:s.

Pure invertibility and overloading of functions in an object-oriented query language are treated in Flodin and Risch [86], and Flodin [87]. However, as shown above, pure object-orientation, is not sufficient in the present domain but overloading an all arguments is required. Hence, the method of Flodin and Risch [86] is not sufficient in this case. Due to this fact, the method has been generalised to handle overloaded multi-directional functions with late binding, Flodin [87] and Flodin et al. [19]. Thus, the present mechanism for selecting the appropriate matrix algebraic operation (i.e. function) is based on type-dispatch for functions overloaded on all argument in combination with late binding. This works fine for the examples in our application since a unique execution plan exist in each case. However, problems can at least theoretically arise when more than one operator is applicable. For this situation, our technique can be extended with a cost-based resolution technique for overloaded functions.

The previous section provided a taxonomy of matrix categories and matrix operators that could be specialised for different combinations of matrix categories. These concepts form the base for the implementation of numerical matrix algebra in the AMOS environment. The conceptualisation of the matrix domain that has been designed and implemented covers numerical matrices of integer, float, and double data types. Included in this conceptualisation are matrix categories and operators that were presented in the former parts of this section. This also involves the regular and skyline representations, but there is currently no sparse representation (mainly due to its absence in the original TRINITAS system). Not every possible variant of the basic operators is currently covered but the functionality exceeds the ability of solving linear equation systems of both regular and skyline matrices. However, it is expected that the basic operators can be combined to form additional algorithms.

Matrix categories are implemented as a type taxonomy with the basic structure according to Figures 23, 24, 25, and 26[1]. There are currently three basic inheritance structures that rely on multiple inheritance to provide the intended functionality to concrete types. The representation of the three numerical data types are currently modelled by explicit types, illustrated in Figure 25. In future developments it would be possible (dependent on execution overhead) to treat these representations implicitly by the use of type templates in a similar way to C++, [146]. Then int, float, and double could share the storage method and the right data type is accomplished implicitly by casting at access time. The type structure that represents mathematical concepts, illustrated in Figure 23, have further been separated from the matrix representation schemes that have their own type structure, illustrated in Figure 24. A similar separation is suggested by Chambers and Leavens [147].

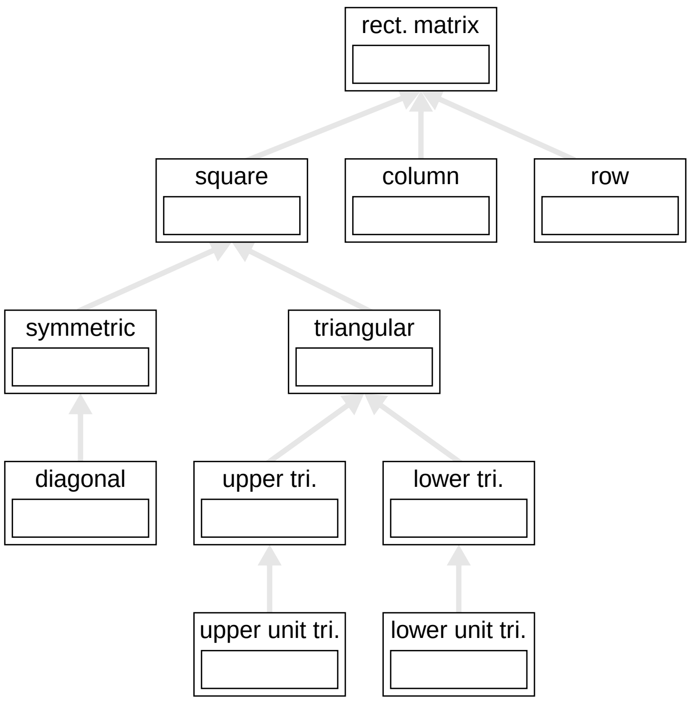


Figure 23. *The current type taxonomy for representing the mathematical matrix categories.*

All matrix types are currently based on one of the numerical array types, described in

1. The type names in the figures have been shortened in some case and the illustration of the type taxonomy has been slightly compacted for facilitating the overview. The complete type taxonomy can be studied in Appendix B.

Section 5.2.4, for their physical representation. They are further implemented as linear arrays, applying a column-order storage sequence. Multiple inheritance is also used for inheriting basic properties from the array types.

As the needs are extended, the type taxonomy can be extended to handle additional representation schemes and data types as indicated for a sparse representation scheme in Figure 24. It should be noted that it might also be suitable to extend the type taxonomy to handle additional mathematical matrix categories as well. For instance, when the domain is extended with additional problem categories it can be necessary to distinguishing between matrices that are positive definite, positive semi-definite, etc. Whether the treatment of additional matrix properties should be accomplished by additional subtypes or by another mechanism must be further investigated.

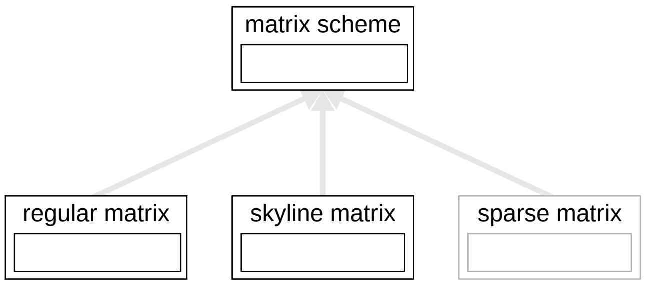


Figure 24. *The type taxonomy for representing numerical matrix representation schemes. Regular and skyline matrix schemes are currently implemented, whereas the sparse scheme indicated are not at present included.*

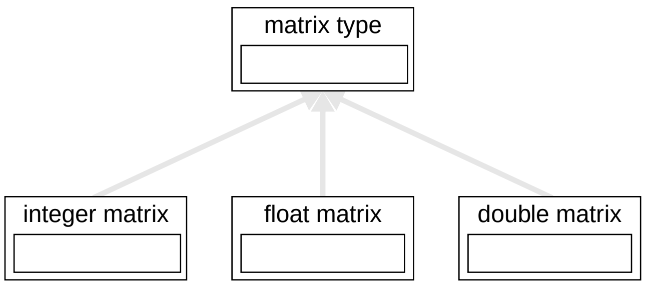


Figure 25. *The type taxonomy for numerical matrix data types where there are subtypes for integer, float, and double data types of numerical data. The concrete representations are inherited from the corresponding numerical*

array type that are currently used for the physical representation of all matrix types.

The properties and operations in the matrix domain are implemented by means of AMOSQL functions. Basic matrix operations that are not specific to matrices are inherited from other supertypes, such as array types. An example is the size function that is defined for the array type. Functions are further multi-directional where it is applicable. The name function can, for instance, be used for retrieving matrix names or for retrieving matrices with a given name. There are also several variants of some functions due to overloading on different function signatures. This is not revealed in the subsequent list of functions. A more complete list of overloaded foreign functions can be found in Appendix C.

- **construct**(charstring *typename*, vector *settings*) -> boolean
- **initialise**(matrix *mat*, vector *settings*) -> boolean
  The initialise function is used for specifying initialisations specific to matrices and is applied by the generic construct function.
- **destruct**(object *obj*) -> boolean
  The destruct function implements specific matrix behaviour that should be applied when a matrix is destructed by the generic destruct function.
- **name**(matrix *mat*) -> charstring *aname*
  The name function is a stored attribute that is used for user-defined naming of matrices.
- **size**(matrix *mat*) -> integer *asize*
  The size function derives the size of the matrix that was set at creation time.
- **rows**(matrix *mat*) -> integer *nrows*
  The number of rows of the matrix is held by the stored function rows.
- **columns**(matrix *mat*) -> integer *nrcols*
  The number of columns of the matrix is held by the stored function columns.
- **ref**(matrix *mat*, integer *indexi*, integer *indexj*) -> number *value*
  Matrix elements can be referenced by the ref function where the second and third argument specify the indexes of the element that should be determined.
- **set**(matrix *mat*, integer *indexi*, integer *indexj*, number *value*) -> boolean
  The set function is used for updating matrix elements.
- **plus**(matrix *m1*, matrix *m2*) -> matrix *m3*
  The plus function is a multi-directional function that implements invertible matrix addition. Alternatively to the prefix notation, the “+” operator can be used in infix notation.
- **minus**(matrix *m1*, matrix *m2*) -> matrix *m3*

The minus function is a multi-directional function that implements invertible matrix subtraction. Alternatively to the prefix notation, the "-" operator can be used in infix notation. Subtraction is defined as the inverse of plus, i.e. the code that implements plus is reused for the minus function.

- **times** (matrix *m1*, matrix *m2*) -> matrix *m3*
  The times function is a multi-directional function that implements invertible matrix multiplication. Alternatively to the prefix notation, the "*" operator can be used in infix notation. Multiplication is overloaded on different function signatures to implement efficient operations and for code reuse.
- **quotient** (matrix *m1*, number *n*) -> matrix *m2*
  The quotient function is a multi-directional function that implements invertible matrix division where applicable. Alternatively to the prefix notation, the "/" operator can be used in infix notation. Currently, the quotient function is not used in the application.

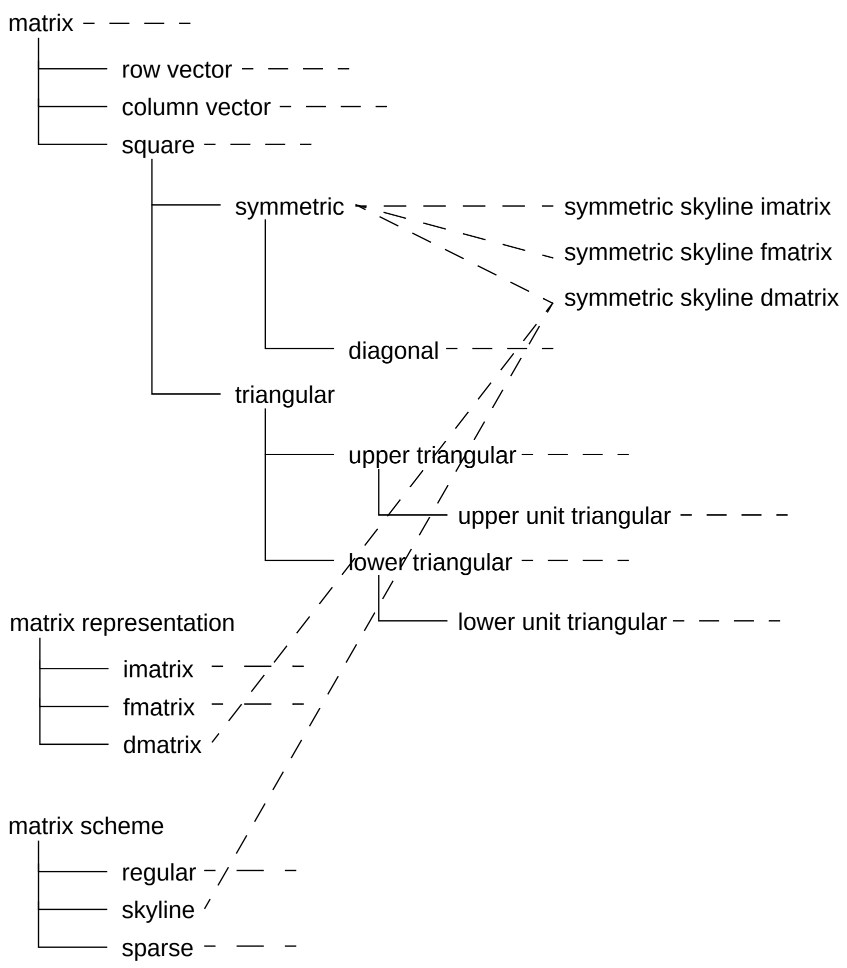


Figure 26. *The composite type taxonomy for the matrix domain with individual inheritance structures for the three basic categorisation principles.*

- **transpose**(matrix *m1*) -> matrix *m2*
  Transposition of matrices that does not do any physical reorganisation, it only creates a new object with a different type without copying data.
- **factorise**(symmetric_matrix *m1*) -> <upper_unit_triangular_matrix *m2*, diagonal_matrix *m3*>
  Factorise implements decomposition of symmetric matrices. Furthermore, it is currently implicitly assumed that matrices are nonsingular. Currently the factorise function implements LDL^T decomposition (or Crout decomposition) that divides a

symmetric matrix C into three matrices L, D, and U, where L * D * U, L = transpose(U), U is an upper unit triangular matrix, D is a diagonal matrix. Here, only D and U are generated.

- **dindex**(skyline_matrix *m*) -> iarray a
  The dindex function stores the diagonal index of a skyline matrix to be used by other operations for locating specific columns.

Other operators can be expressed in terms of these basic matrix operators, and of course, the current set of base operators can be extended. As an example of the former, a linear combination of matrices can easily be expressed using the AMOSQL infix notation as,

```
select m3 for each matrix m3
                    where :a * m1 + :b * m2 = m3;
```

where `:a` and `:b` are reals and `:m1` and `:m2` are matrices, or by function composition using prefix notation:

```
select m3 for each matrix m3
                    where plus(times(:a,:m1), times(:b,:m2)) = m3;
```

This expressibility is accomplished by overloading the times operator on combinations of number and matrix. Further, expressing this as a function would look like

```
create function linear_combination(real a, matrix m1,
                                   real b, matrix m2) -> matrix as
                    select a * m1 + b * m2;
```

5.2.4 The array foreign data source

Many scientific and engineering applications involve one- or multi-dimensional sequences of numerical data expressing arrays or matrices of numerical values. These are usually used to represent different mathematical concepts, such as scalar-, vector-, or dyadic-valued quantities. Thus, the ability to represent and do computations on sequences of numerical data is of great importance for scientific and engineering software tools. Several commercial “tool kits” for scientific and engineering computations, like MATLAB[1], MATHCAD[2], and HiQ[3], support these facts.

Likewise, the database community has emphasised the ability to represent sequences of numerical data. This capability can, for instance, be found in commercial products like Illustra [6] and eBASE[4]. It has also been developed for research DBMSs including EX-

1. MATLAB is a product of MathWorks, Inc.
2. Mathcad is a product of MathSoft, Inc.
3. HiQ is a product of National Instruments, Inc.
4. eBASE is a product of Universal Analytics, Inc.

ODUS [99]. Furthermore, the standard proposals of SQL3 [16] and OQL [17] include data structures for numerical sequences in their object model. By providing data structures for numerical sequences in a database environment it will be possible to combine this type of numerical data with other data types, such as simple integer and real number data, character strings. Hence, an extended set of data structures are made available for facilitating modelling and manipulation of the rich set of scientific and engineering data.

In this work, the AMOS object-relational DBMS has been extended with a numerical array data structure. At the query language level, three different numerical and one-dimensional, or linear, array types have currently been defined and implemented. The array types (or classes) are defined for sequences of integers, floats, and doubles, that are denoted:

- iarray
- farray
- darray

where iarray, farray, and darray represents a fixed sequence of 4-byte integer numbers, 4-byte real numbers, and 8-byte real numbers respectively. These three types are formed in a type structure according to Figure 27 where the array type is a subtype of the usertypeobject type.

Eight basic operations have currently been defined for arrays at the query language level:

- **construct**(charstring *typename*, vector *settings*) -> boolean
- **initialise**(array *arr*, vector *settings*) -> boolean
- **destruct**(array *arr*) -> boolean
- **name**(array *arr*) -> charstring *aname*
- **size**(array *arr*) -> integer *asize*
- **ref**(array *arr*, integer *index*) -> number *value*
- **set**(array *arr*, integer *index*, number *value*) -> boolean

where name is a stored function representing an optional name attribute and size is a foreign function that represents the length of the array. The construct and destruct operations are general constructor and destructor procedures for implementation of tailored creation and deletion operations. The construct operator takes a name of a type and a vector of initial settings and creates an object of that type. It then applies the initialise operator, overloaded for different types, that defines specific initialisations for the created object. Finally, the construct operation returns the newly created and initialised object.

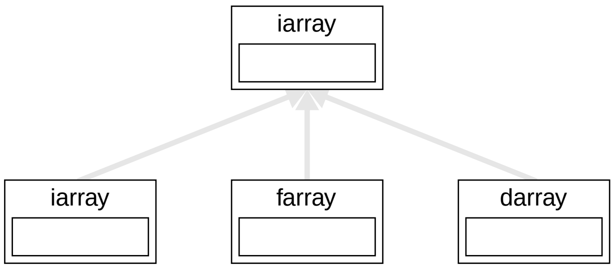


Figure 27. *The type hierarchy for numerical arrays where the array type is a subtype of usertypeobject.*

The current implementations of the initialise and the construct functions for objects are as follows:

```
create function initialise( object obj, type tp,
                           vector settings) -> object as
                    begin
                       /* type-specific initialisations */
                       result obj
                    end;

create function initialise( array arr, type arrtype,
                          vector settings) -> array as
                    begin
                       allocate(arr, arrtype, settings);
                       set name(arr) = vref(settings, 1);
                       result arr
                    end;

create function construct( charstring typename,
                           vector settings) -> object obj as
                    begin
                       declare type tp;
                       set tp = typenamed(typename);
                       set obj = new_object(tp);
                       initialise(obj, tp, settings);
                       result obj
                    end;
```

As an example of how tailored constructor functionality can be accomplished, the second initialise function above is designed for the array type by overloading the initialise function for a corresponding array type signature. The values of the size and name attributes are extracted from the settings vector and initiated by the allocate and set operations. More specifically, the allocate operation is responsible for allocating a literal array object (a basic data storage structure) of a specified size and associating it with an object of type array. The constructor and destructor operations were mainly introduced to provide tailored creation and deletion for array types and subsequent extensions to matrices and domain-specific types presented in Section 5.3.3. Parallel work on AMOS has generalised the applicability of constructors to any user-defined type [93]. Similar functionality is specified in the SQL3 proposal. The ref and the set operator are foreign functions for retrieving and updating single array values. Additional retrieval and update operators are required as well, but these are still provided by the FEA application. This includes operators for accessing subparts of an array and similar operations on other aggregation data structures based on arrays. Operations of this type are not normally supported in array representations for databases but are of great importance to engineering applications and are supplied in "tool-kits" such as MATLAB and others. As we have seen in the previous section, this basic array type structure is further extended by adding subtypes of arrays and by adding operators that are required in specific applications.

Below the query language level, the new array data source is defined by a literal array data type. The array data type is complemented by a set of basic operations that are accessible from both LISP and C. Foreign functions, defined at the query language level, are defined by means of operations at this level that operate on literal arrays. Since it is possible to dereference the array data structure from the literal object, it is possible to implement critical and kernel array operations as efficiently as in conventional programming languages.

Hence, as described in Section 4.4, C record templates for the literal array data types are defined for iarray, farray, and darray. The techniques for defining and registering new storage data types that were described in the same section have been used. An example of the record template for the farray data type (float array) is provided below.

```
struct farray
{  objtags tags;
   char filler1[2];
   int len;
   char filler2[4];
   char cont[sizeof(float)];};
```

The `objtags` include type information and reference counters for storage management, `len` is the size of the array, and the last. `char` declaration is a pointer to the data segment. The basic data structure is of the same type as `arrays` in C and Fortran.

There are currently five basic operations defined on literal arrays which are implemented in C. Here, they are exemplified for float arrays:

- **mkfarrayfn**(arraysize) creates a literal float array object where the number of elements is determined by arraysize.

- **farraysizefn**(array) retrieves the size of the literal array object.
- **farrayreffn**(array, arrayel) retrieves the value of the float, in the literal array object, at the position determined by index.
- **setfarrayfn**(array, arrayel, elvalue) assigns the float value to the float, in the literal array object, at the position determined by index

These operations are all implemented in C. They are also registered to LISP using a standardised naming convention and are accessed as **mkfarray**, **farraysize**, **farrayref**, and **setfarray**.

There is an additional operation:

- **floatcont**(farray-oid) that returns the dereferenced array of floats (the basic array data storage structure) of the literal array object.

The floatcont operation is a C-macro and is a low-level operation that should only be used within other operations to be able operate on and index the basic array data structure.

The numerical array data structures can be further developed to include dynamic arrays and maybe multi-dimensional (at least two-dimensional) arrays. For numerical analysis applications, you usually use some form of tailored two-dimensional representation of arrays that takes advantage of domain-specific characteristics to provide more compact representations of multi-dimensional arrays, e.g. skyline matrix representations. This is one major reason behind the decision to use this array representation to implement the matrix representation as discussed in the previous section. The performance for the current array representation is discussed in a special section, Section 5.4, that treats related performance issues as well.

5.3 FINITE ELEMENT ANALYSIS DOMAIN MODELLING

Domain modelling involves the actual conceptualisation of the FEA domain into concepts, relationships, and operations in a high-level and problem-related terminology. A suitable conceptual level and data representation should be tuned to processing needs. This usually requires that a number of design iterations are performed. Increasing the conceptual level in the analysis, design and implementation of engineering applications is an important principle for developing these applications more efficiently. A higher level of representation independency and resistance can be obtained if the conceptualisation is based on theories and basic principles instead of on application-specific representations. Further, logical descriptions and representations (such as database schemas) of a domain are more general and reusable than actual physical implementations. As already discussed, query language technology provides the application developer with modelling capabilities at a higher level than that of a conventional programming language level.

The FEA domain has coarsely been divided into a number of subdomains that covers concepts related to geometry, boundary, domain, finite elements, and linear matrix algebra. This division is similar to the general structure within the associated TRINITAS FEA system. However, there are also differences. Since TRINITAS is in some sense “object-based”, i.e. it is structured around FEA-related abstract data types, it is not really an object-oriented system. The current design, within AMOS, has introduced a more object-oriented view on the FEA domain using, for instance, inheritance, encapsulation, and function overloading to structure the problem domain. There are furthermore, some intermediate design trade-offs in the system that are related to the merging of the original “object-based” and the new object-oriented conceptualisation of the FEA domain. These trade-offs include mappings between OID:s and names, location of operations and index utilization, and unsuitable data representations. However, these are all temporary problems that could be dealt with in further development. These design flaws must be accepted in this intermediate stage, where an FEA domain model is not completely established that takes advantage of potential database capabilities. Likewise, to what extent the object-oriented design should be performed must be further examined, i.e. which kinds of concepts are suitable to represent as objects and to what level of detail should the object-orientation be taken. These issues will be exemplified in subsequent sections.

Currently, the conceptualisation mainly covers geometry-, analysis-, and matrix-related concepts. Examples also show how concepts related to discretisation and calculated results can be modelled and manipulated by the query language. Boundary- and domain-related concepts, such as boundary conditions and material models, are not covered in the present work and remain to be addressed in future work.

5.3.1 Geometry and topology

The geometry is the base for specifying FEA problems in TRINITAS, similar to the approaches presented in Myers [66] and Finnigan et al. [148]. The complete problem is specified with respect to the geometry including loadings, other boundary conditions and domain conditions. In FEAMOS, the geometric model is a transformation of the geometric concepts in TRINITAS into a corresponding database schema in AMOS. TRINITAS builds up the geometry from basic entities of various geometric categories such as points, curves, surfaces, and volumes as listed in Appendix A. Categories such as straight line, arc, triangular surface, and quadrangular surface are represented by named and fix-sized arrays in TRINITAS. The TRINITAS geometric categories are transformed into a class taxonomy with the basic structure illustrated in Figure 28. An abstract class `geometry_object` holds the basic geometry classes `volume, surface, curve,` and `point`. The complete class taxonomy includes several subclasses and is presented in Figure 30. For instance, the curve class has the subclasses straight line, arc, parabola cubic section, and Bezier cubic segment. The class taxonomy is modelled by a type and subtype structure in AMOSQL.

Topology relations, i.e. relations among geometric entities such as `faces`, `edges`, and `vertices`, are modelled as functions between basic geometry classes as illustrated in Figure 29. The forks at the end of the relations indicate many-to-many relationships, i.e. for instance that a surface can have many edges and each curve can be an edge to many surfaces. Since values of normal and multi-valued stored functions in AMOSQL are unordered, the topology functions are represented by vectors to accomplish an ordering of vertices, edges, etc. For example, the function `vertex_vector` is defined by:

```
create function vertex_vector(curve c) -> vector as stored;
```

In addition, a derived `vertices` function have been defined for accessing each element more conveniently in ad hoc queries[1]:

```
create function vertices(curve c) -> point p as
                select element(vertex_vector(c));
```

The `vertices` function provides a relation between a curve instance to its defining points with the intuitive direction from the curve to the points, but with applicability in both directions. Spatial coordinate data is defined for the `point` class where the `position` function stores the x, y, and z coordinates of a point instance. Coordinates can also be retrieved by the derived functions x_coordinate, y_coordinate, and z_coordinate. The `position` function is currently implemented as a tuple of three reals, which causes unnecessary data conversions in manipulating these data. The implementation of the FEAMOS geometry model uses the previously described matrix package in its implementation in AMOS. This has made it possible to use database representations suitable for numerical processing, and replacing the tuple representation of the coordinates with a matrix representation eliminates the need for data conversions and facilitates numerical processing of coordinate data.

In addition, to keep an intermediate correspondence with TRINITAS, a number of attributes are currently defined for different geometric categories but might later suit better in other concept classes. These attributes are located below the dashed lines in Figure 28. There are no absolute truths on how to locate attributes to specific concepts and this issue will not be discussed in any depth. A good conceptualisation is most certainly based on knowledge about the domain, experience in domain modelling, and an iterative evolution of the design. As an example, the `curve` type currently has a `division` and a `density` function implemented. The `division` function represents the number of subdivisions a specific curve is divided into and the `density` function represents a node density along a curve. Further studies might reveal that it would be more convenient to associate these attributes to a mesh concept related to the curve since the

1. It should be noted that an extension of AMOSQL to handle ordered sets, or sequences, can eliminate the need for the `vertices function.`

division and density are, in fact, attributes related to the discretisation of a geometry. However, other issues, such as model complexity and processing efficiency, can influence the outcome of these decisions as well.

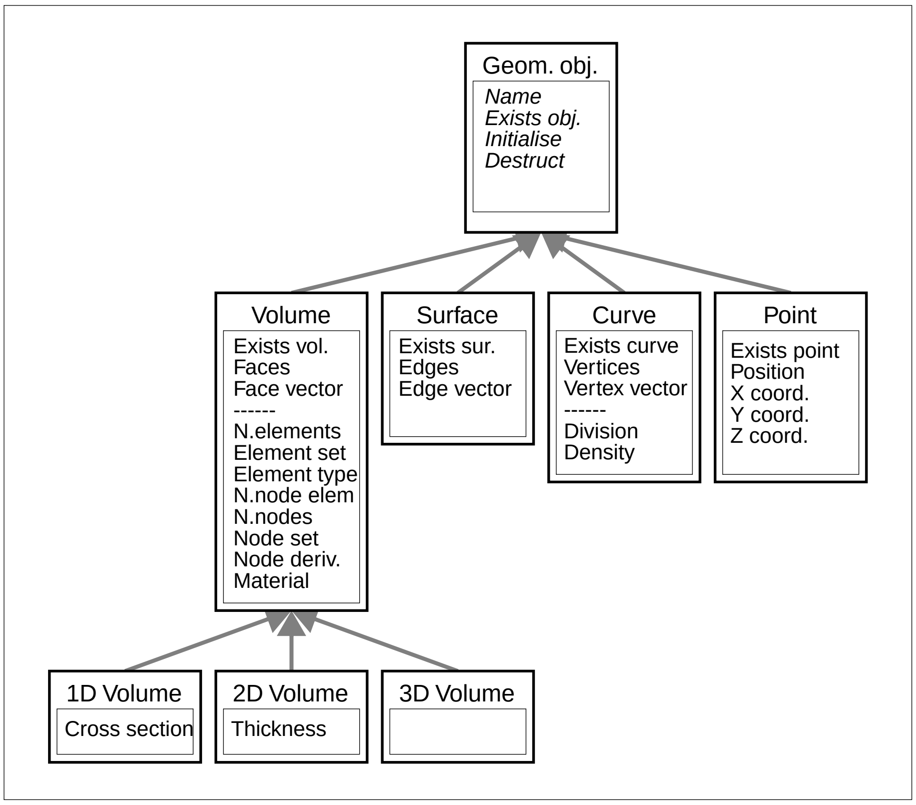


Figure 28. *Basic type taxonomy for geometry-related concepts in the application domain, where arrows denote is-a (subtype to supertype) relations.*

Even if the geometric schema includes those geometric concept classes of TRINITAS, the geometric schema is designed in a general manner to be applicable in other applications as well. There is no application-specific information represented in the basic geometric model. Where applicable, relations are established with object identifiers (OID:s) instead of using names and naming conventions. The geometric model can easily be evolved with new geometric concepts or the schema can be changed without affecting the application. The application communicates with the general geometry model through a set of derived functions that form a view of the geometry model that can be tailored for a specific application. For example, TRINITAS currently uses application-generated names to refer to "objects". In the geometry model of FEAMOS, these names are not used for references but they are stored as redundant information in the database and used for interfacing between this database and the application.

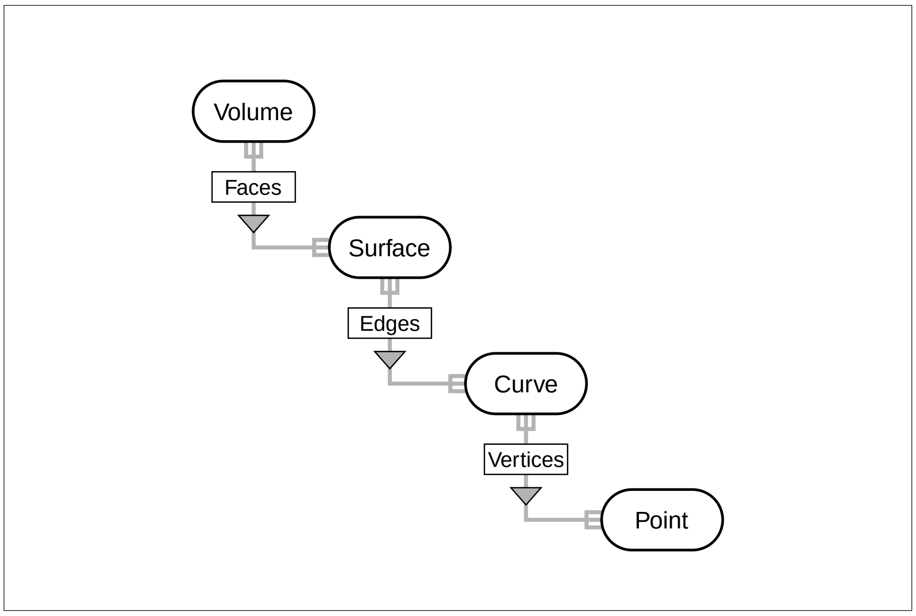


Figure 29. *Topological relations (rectangular symbols) between basic geometry concepts with the intuitive direction indicated by triangular arrowheads.*

Operations on geometrical objects are implemented by means of AMOSQL functions and procedures. TRINITAS operations can be divided into basic operations and composite operations. Basic operations include creation and deletion of geometric entities as well as operations for accessing and changing their properties. Composite operations are more complex and usually more domain-oriented operations such as finding specific sets of geometric entities, or calculating geometry-dependent quantities.

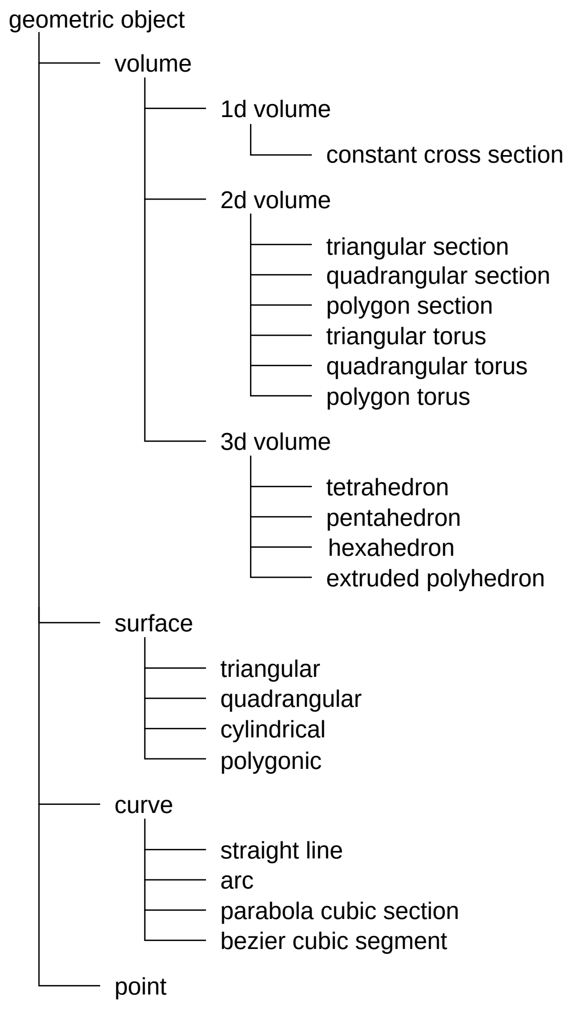


Figure 30. *The current type taxonomy for the geometric model in FEAMOS.*

Since the composite operations partly include basic operations, this initial representation of the geometry model within the database has focused on including the set of basic operations that can be seen as the actual application interface to the geometry model. When more and more functionality is transferred to the DBMS, this interface will grow to incorporate more domain-specific operations. Hence, there are database operations implemented for constructing and destructing geometry objects. For this purpose, it would be convenient to be able to define tailored constructors and destructors that can be overloaded. The AMOS system did not permit this kind of definition at the time when the geometry model was implemented. However, this was later implemented to be used for matrices, described in Section 5.2.4. Later again, this functionality has been generalised by Werner [93] and implemented in AMOS. Currently, the geometry model still uses specialised procedures for creating points, straight lines, etc., but these can later easily be replaced by overloaded constructors. Furthermore, the access functions and the update procedures have been implemented as derived AMOSQL functions in terms of basic stored functions. In this manner, the application interface forms a view to the database schema supporting a data-independent design.

In FEAMOS, when a geometry is modelled in TRINITAS, an object structure is generated in an AMOS database. The original FORTRAN procedures that handle geometry in TRINITAS have been redesigned to use the AMOS C interface procedures to communicate with the DBMS. There is no redundant representation of the application model, it exists only within the database. Thus, when an object is manipulated, as for instance moving a point in a geometry model on the screen, it implies a direct update of the database object.

The modelling of a geometry can be illustrated by means of the example shown in Figure 31. The picture shows a rectangular plate that is fixed on a wall at its left side and is further exposed to a distributed and uniform pressure load at the upper edge. In this case, the geometry model is formed by a number of basic geometric elements, i.e. 4 points, 4 straight lines, 1 surface element, and 1 volume element, shown in Figure 32.

This can be expressed in AMOSQL[1] as:

```
create point(name, position)
          :p1("p1", <0.0, 0.0, 0.0>),
          :p2("p2", <0.0, 120.0, 0.0>),
          :p3("p3", <90.0, 120.0, 0.0>),
          :p4("p4", <90.0, 0.0, 0.0>);

create straight_line(name, vertex_vector, division, density)
          :l1("l1", {:p1, :p2}, 3.0, 0.0),
          :l2("l2", {:p2, :p3}, 3.0, 0.0),
          :l3("l3", {:p3, :p4}, 3.0, 0.0),
          :l4("l4", {:p4, :p1}, 3.0, 0.0);

create quadrangular_surface(name, edge_vector)
          :s1("s1",{:l1, :l2, :l3, :l4});
```

1. To facilitate the interpretation, an interactive style of programming is used the example in contrast with using the application programming interface for expressing the same functionality.

```
create quadrangular_section_volume(name, face_vector)
          :v1(“v1”, {:s1});
```

These statements create a geometric model where the geometric objects are structured as illustrated in Figure 33. Through TRINITAS, the model is built up graphically and the result is illustrated in Figure 34.

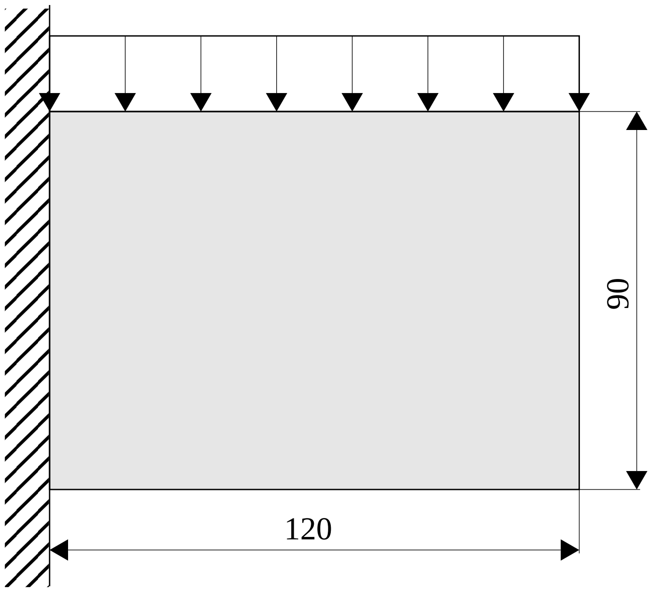


Figure 31. *A simple physical structure consisting of a fixed plate that is exposed to a distributed and uniform pressure load.*

The AMOSQL query language can then be used to formulate queries to the model about its structure and content, such as basic geometrical and topological information. For example the edges of `:s1` can be extracted by:

```
edges(:s1);
OID[0x0:765] OID[0x0:766] OID[0x0:767] OID[0x0:764]
```

Similarly, the edges can be extracted from `:v1 by:`

```
edges(faces(:v1));
```

```
OID[0x0:765] OID[0x0:766] OID[0x0:767] OID[0x0:764]
```

Actually, to be sure that an edge will only occur once if it is of several surfaces that form a volume, the query should be reformulated, using the `unique`[1] function, as:

```
unique(edges(faces(:v1)));
```

If it is common to access the points from other levels than at the curve level, it can be convenient to overload the `vertices` function as a derived function on other geometric classes as well:

```
create function vertices(volume v) -> point p as
                    select p for each curve c
                       where  p = unique(vertices(c)) and
                              c = unique(edges(faces(v)));
```

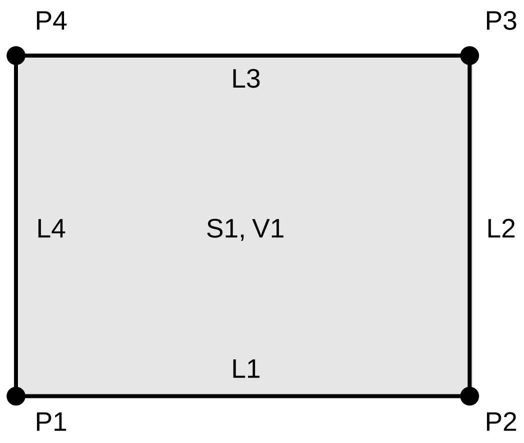


Figure 32. *Basic geometrical model of the structure in Figure 31 including boundary conditions.*

By applying the `edges` function in the opposite direction, we can find the surface that a specific curve is a part of:

1. The `unique` function is an aggregation function that eliminates duplicates from a multiset.

```
select s for each surface s where edges(s) = :l2;
OID[0x0:768]
```

The `edges` function is modelled to store object identifiers internally for generality and efficiency reasons. However, the `name` function can also be applied on each object for name reference. The first of the preceding examples would then look like:

```
name(edges(:s1));
“L2“ “L3“ “L4“ “L1“
```

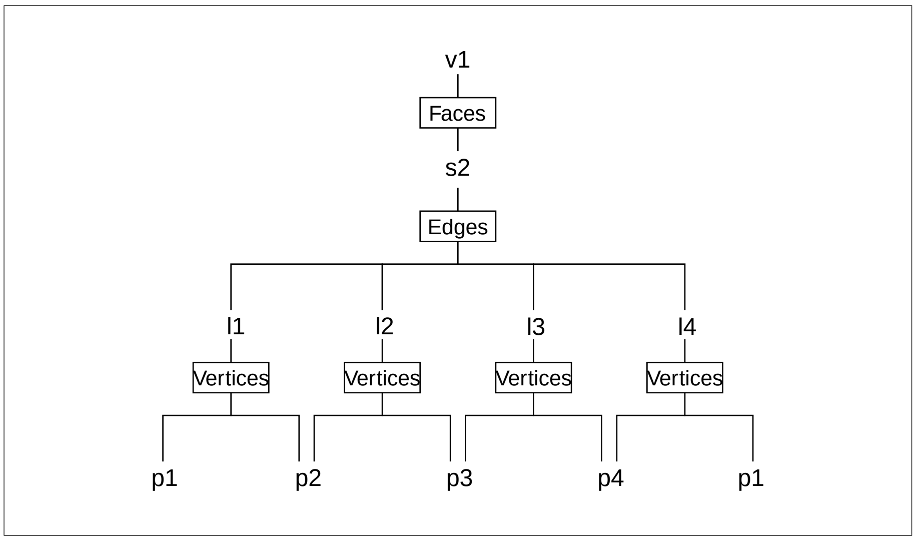


Figure 33. *The structure of an instance model of a simple geometric model in FEAMOS.*

In addition to basic operations for managing data, the application includes more complex and composite operations that form the domain functionality of the application. As argued at the beginning of this section, there are several factors that influence the decision of where to place operations. Additionally, it might imply a severe redesign of the application logic when operations implemented in a conventional and procedural programming language should be expressed in a database language that is of a more declarative nature. Geometric information is involved in many tasks within FEA that are spread all around the application. A redesign at the global level has currently not been considered within FEAMOS, that covers the complete use of geometry information. Nevertheless, a few examples will show how AMOSQL can be used to model more complex and composite geometric operations.

Operations on geometry to find different sets of boundary objects of the geometry occurring frequently. For instance, TRINITAS includes an operation for retrieving the corners of different geometric entities. In AMOSQL, this is expressed for a quadrangular section in the `corners` function as:

```
create function corners(quadrangular_surface qs) -> point as
                    select p1 for each curve c1, curve c2, point p1
                       where  c1 = edges(qs) and
                              c2 = edges(qs) and
                              c1 < c2 and
                              p1 = vertices(c1) and
                              p1 = vertices(c2);
```

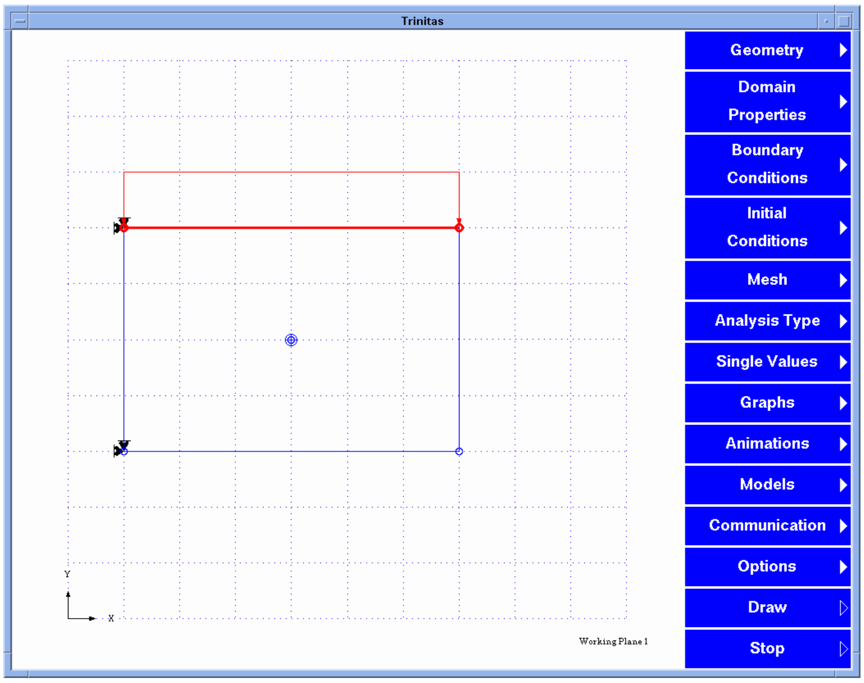


Figure 34. *The geometry model example modelled in TRINITAS completed with boundary conditions in the form of a fixed boundary and a distributed and uniform load.*

Currently, AMOSQL mainly support bag-valued functions. This means that functions in general return multisets and for obtaining sets the application of additional filtering functions is required. In, for instance, the `corners` function this has been avoided by reducing the qualifying data sets by replacing the `!=` (not equal to) function by the `<` (less than) function. There is a common interest in engineering applications to be able to handle sets and ordered sets (or sequences), in addition to multisets. Although AMOSQL supports data types for sets and sequences, queries can currently not automatically handle and preserve ordering and duplication elimination. A support for handling sequences and sets in queries will facilitate the specification of common application-specific operations where unique and ordered results are desirable. Future incorporation of this functionality in AMOSQL is being considered.

Applying the corner function to our geometry example should return the four corner points of our model as:

```
select name(corners(s)) for each surface s;
“P3“ “P4“ “P2“ “P1“
```

The compact expressiveness of AMOSQL is illustrated by a comparison with the corresponding `GET_SURFACE_CORNER_POINTS` procedure in TRINITAS:

```
SUBROUTINE GET_SURFACE_CORNER_POINTS(CORNERS,LINES,
                                     UNIQUE_POINTS)
       INTEGER CORNERS,TYPE,I,J,K,NHIT
       CHARACTER LINES(1)*8,POINTS(4)*8,UNIQUE_POINTS(1)*8,
   %   END_POINTS(2)*8
       NHIT = 0
       DO 10 I=1,CORNERS
          CALL GET_LINE(LINES(I),TYPE,POINTS)
          CALL GET_LINE_END_POINTS(TYPE,POINTS,END_POINTS)
          DO 20 J=1,2
             DO 30 K=1,NHIT
                IF(UNIQUE_POINTS(K).EQ.END_POINTS(J)) GOTO 20
   30        CONTINUE
             NHIT = NHIT + 1
             UNIQUE_POINTS(NHIT) = END_POINTS(J)
   20     CONTINUE
   10  CONTINUE
       END
```

In the TRINITAS procedure the specific surface is implicitly defined by its edges that are passed as the `LINES` argument. Here is also an additional argument, `CORNERS`, that must be bound before calling the procedure as exemplified in the `GET_SURFACE_CG` procedure below.

```
SUBROUTINE GET_SURFACE_CG(SURFACE_NAME,TYPE,LINES,TP)
       INTEGER SC(3),TP(2)
       INTEGER TYPE,I,CORNERS,GET_NO_SURFACE_LINES
       REAL*4 GC(3),GC0(3)
       CHARACTER LINES(1)*8,POINTS(20)*8,SURFACE_NAME*8
```

```
C   Get number of surface corners
       CORNERS = GET_NO_SURFACE_LINES(SURFACE_NAME,TYPE)
C   Get all unique corner points
       CALL GET_SURFACE_CORNER_POINTS(CORNERS,LINES,POINTS)
C   Calculate the centre of gravity
       GC0(1) = 0.
       GC0(2) = 0.
       GC0(3) = 0.
       DO 10 I=1,CORNERS
          CALL GET_POINT(POINTS(I),GC)
          GC0(1) = GC0(1) + GC(1)
          GC0(2) = GC0(2) + GC(2)
          GC0(3) = GC0(3) + GC(3)
    10 CONTINUE
       GC0(1) = GC0(1)/CORNERS
       GC0(2) = GC0(2)/CORNERS
       GC0(3) = GC0(3)/CORNERS
       CALL GET_SCREEN_COORDINATE(GC0,SC)
       TP(1) = SC(1)
       TP(2) = SC(2)
       END
```

Similar observations can be made, as in the previous example, when comparing with the corresponding functionality expressed in the AMOSQL `centroid`[1] function:

```
create function centroid(quadrangular_surface qs) ->
                         <real x,real y,real z> as
                    select x,y,z
                       where
                          x = mean(x_coordinate(corners(qs))) and
                          y = mean(y_coordinate(corners(qs))) and
                          z = mean(z_coordinate(corners(qs)));
```

where `mean` is an aggregation function that calculates the numerical mean value of its argument. An application of `centroid` to our example looks like:

```
centroid(OID[0x0:768]);
<40.,50.,0.>
```

An additional example shows an AMOSQL function, `opposing_curves`, for extracting pairs of curves that are opposite for a quadrangular surface.

```
create function opposing_curves(quadrangular_surface qs) ->
                                <curve c1, curve c2> as
                    select unique((select c1,c2
```

1. It should be noted that for a general quadrangular, a more advanced calculation of the centroid is required. This functionality is currently absent in TRINITAS.

```
                for each curve c1, curve c2
                   where
                      c1 = edges(qs) and
                      c2 = edges(qs) and
                      notany(end_points(c1) =
                      end_points(c2))));
```

Applying the `opposing_curves` function on the quadrangular surface in our example gives us the set of tuples of related curves, such as:

```
opposing_curves(OID[0x0:768]);
```

```
<OID[0x0:765] ,OID[0x0:767] >
<OID[0x0:766] ,OID[0x0:764] >
<OID[0x0:767] ,OID[0x0:765] >
<OID[0x0:764] ,OID[0x0:766] >
```

Note that this function can also be used for extracting the opposing curve to a specific curve of a surface as in the following query:

```
select unique((select c for each curve c
         where
            opposing_curves(OID[0x0:768]) = <c, OID[0x0:766]>));
```

```
OID[0x0:765] OID[0x0:767] OID[0x0:764]
```

Again, comparing with TRINITAS, the `ADD_OPPOSITE_AND_RELATED_LINES` subroutine include a similar functionality as the `opposing_curves` function. However, one branch of the Fortran subroutine treats triangular surfaces whereas another treats quadrangular surfaces. In AMOSQL, this would be accomplished by overloading the `opposing_curves` function on triangular_surface as well.

```
SUBROUTINE ADD_OPPOSITE_AND_RELATED_LINES(SURFACE_NAME,
   %         LINE_NAME,CURRENT_NO_SUBDIVISIONS,
   %         LINE_LIST, NUMBER_OF_LIST_MEMBERS)
      INTEGER NUMBER_OF_LIST_MEMBERS,TYPE,I
      INTEGER CURRENT_NO_SUBDIVISIONS
      REAL*4 ATTRIBUTE(4)
      CHARACTER SURFACE_NAME*8, LINE_NAME*8, LINE_LIST(1)*8
      CHARACTER LINES(20)*8, POINTS(4)*8, END_POINTS(2)*8
      CHARACTER OTHER_END_POINTS(2)*8
      LOGICAL   GET_SURFACE,GET_LINE_ATTRIBUTES,LINE_ON_LIST
      IF(GET_SURFACE(SURFACE_NAME,TYPE,LINES)) THEN
        IF(TYPE.EQ.1) THEN
          DO 10 I=1,3
            IF(LINES(I).NE.LINE_NAME) THEN
           IF(GET_LINE_ATTRIBUTES(LINES(I),
   %                     ATTRIBUTE)) CONTINUE
              IF(.NOT.LINE_ON_LIST(LINES(I),LINE_LIST,
   %             NUMBER_OF_LIST_MEMBERS).AND.
   %               INT(ATTRIBUTE(1)).NE.CURRENT_NO_SUBDIVISIONS)
                 THEN
                   NUMBER_OF_LIST_MEMBERS =
   %               NUMBER_OF_LIST_MEMBERS + 1
                   LINE_LIST(NUMBER_OF_LIST_MEMBERS) = LINES(I)
                 END IF
             END IF
   10      CONTINUE
         ELSE IF(TYPE.EQ.2) THEN
           CALL GET_LINE(LINE_NAME,TYPE,POINTS)
           CALL GET_LINE_END_POINTS(TYPE,POINTS,END_POINTS)
             DO 20 I=1,4
               CALL GET_LINE(LINES(I),TYPE,POINTS)
               CALL GET_LINE_END_POINTS(TYPE,POINTS,
   %                                   OTHER_END_POINTS)
               IF(END_POINTS(1).NE.OTHER_END_POINTS(1).AND.
   %              END_POINTS(1).NE.OTHER_END_POINTS(2).AND.
   %              END_POINTS(2).NE.OTHER_END_POINTS(1).AND.
   %              END_POINTS(2).NE.OTHER_END_POINTS(2)) THEN
                 IF(GET_LINE_ATTRIBUTES(LINES(I),
   %                              ATTRIBUTE)) CONTINUE
                   IF(.NOT.LINE_ON_LIST(LINES(I),LINE_LIST,
   %                 NUMBER_OF_LIST_MEMBERS).AND.
   %                 INT(ATTRIBUTE(1)).NE.CURRENT_NO_SUBDIVISIONS)
                   THEN
                     NUMBER_OF_LIST_MEMBERS=NUMBER_OF_LIST_MEMBERS+1
                     LINE_LIST(NUMBER_OF_LIST_MEMBERS) = LINES(I)
                   END IF
              END IF
   20     CONTINUE
       END IF
       END IF
       END
```

The preceding examples of operations on the geometry mainly considered the retrieval of topologic information, but also included some calculation of geometric information, in terms of the centre of gravity, in the centroid function. Another geometric operation, is to find the nearest point to a given point (or position), by calculating the minimal distance. In TRINITAS, this is accomplished by the `FIND_NEAREST_POINT` procedure. The distance calculation is carried out in the `GET_SCREEN_DISTANCE` procedure.

```
LOGICAL FUNCTION FIND_NEAREST_POINT(C,PICKED_POINTS,
   %                              NO_PICKED_POINTS,
   %                              NEAREST_POINT_NAME,
   %                              SCREEN_COORDINATE)
       INTEGER   SCREEN_COORDINATE(3),C(1),SC(3)
       INTEGER   NPOINTS,I,NO_PICKED_POINTS,J
       REAL*4    DISTANCE,GET_SCREEN_DISTANCE,MIN_DISTANCE,GC(3)
       LOGICAL   GET_POINT
       CHARACTER PICKED_POINTS(1)*8,POINT_NAME*8
       CHARACTERNEAREST_POINT_NAME*8
       COMMON    /POINT1/ NPOINTS
       FIND_NEAREST_POINT = .FALSE.
       MIN_DISTANCE = 1.0E38
C   Loop over all defined points
       DO 10 I=1,NPOINTS
          CALL GET_POINT_NAME(I,POINT_NAME)
          IF(GET_POINT(POINT_NAME,GC)) THEN
             CALL GET_SCREEN_COORDINATE(GC,SC)
             DISTANCE = GET_SCREEN_DISTANCE(SC,C)
             IF(DISTANCE.LT.MIN_DISTANCE) THEN
                DO 20 J=1,NO_PICKED_POINTS
                   IF(PICKED_POINTS(J).EQ.POINT_NAME) GOTO 10
   20           CONTINUE
                MIN_DISTANCE = DISTANCE
                NEAREST_POINT_NAME = POINT_NAME
                SCREEN_COORDINATE(1) = SC(1)
                SCREEN_COORDINATE(2) = SC(2)
                SCREEN_COORDINATE(3) = INT(0)
                FIND_NEAREST_POINT = .TRUE.
             END IF
          END IF
   10  CONTINUE
       END

REAL*4 FUNCTION GET_SCREEN_DISTANCE(P1,P2)
       INTEGER P1(1),P2(1)
       GET_SCREEN_DISTANCE = SQRT(REAL(P1(1)-P2(1))**2 +
   %   REAL(P1(2)-P2(2))**2)
       END
```

If we ignore that the TRINITAS procedures involve a transformation of coordinates in space to the screen plane, the same functionality can be expressed by the two AMOSQL functions called `nearest_point` and `distance`.

```
create function nearest_point(point p1) -> point p2 as
                select p2
                   where  distance(p1, p2) =
                          minagg((select distance(p1, p3)
                                  for each point p3));


create function distance(point p1, point p2) -> real d as
                select d for each real x1, real y1,
                                  real x2, real y2
                   where x1 = x_coordinate(p1) and
                         y1 = y_coordinate(p1) and
                         x2 = x_coordinate(p2) and
```

```
                              y2 = y_coordinate(p2) and
                              d = sqrt((x2*x2-x1*x1) + (y2*y2-y1*y1));
```

The application of the `nearest_point` and the `distance` functions, by means of an additional test point :pa, in our example looks like:

```
create point(name, position) :pa(“pa”, <100.0, 100.0, 0.0>);

name(nearest_point(:pa));
<“P3”>

distance(:pa, :p3);
<42.4264>

select name(p),distance for each point p, real distance
          where nearest_point(:pa) = p and
                distance = distance(:pa, p);
<“P3”,42.4264>
```

It should be noted that in the general three-dimensional case the `distance` and the `nearest_point` function can be overloaded on lines instead of points. A screen position can then be viewed as a straight line perpendicular to the screen which makes it possible to calculate and compare the distances in terms of the actual point coordinates of the geometry model. This opens the possibility of using *spatial indexing* techniques to make the search in the point space more efficient. As the set of points in the point space becomes large, spatial indexing techniques can dramatically reduce search effort. However, this issue must be studied in more detail before any specific conclusions can be drawn. There are other FEA data sets as well where spatial indexing can be advantageous. This includes the discretisation of the geometry into mesh data and calculation of quantities, such as stress and displacement fields, distributed over the geometry. Hence, the potential benefit of spatial indexing for different FEA tasks must be evaluated to avoid sub-optimizations.

Several spatial indexing techniques are based on a tree representation where the access cost is dependent on the depth of the tree. An approximated measure of the access cost, C_a, can be expressed in terms of the indexed data set *N;* we get $C_a \sim \log N$. This cost should be compared to the current access cost using linear search (i.e. no indexes) is proportional to *N*.

There is currently no spatial indexing technique available in AMOS and a future research area would be to investigate how spatial indexes could make spatially-related queries more efficient. To be able to make relevant conclusions, this work should probably include comparisons on how indexing, filtering, as well as the location of data and processing influence the total processing costs for different operations, as pointed out in Section 5.1.

5.3.2 The discretisation process

The discretisation process is the intermediate step between the problem specification, in terms of geometry and domain and boundary conditions, and the solution process. The discretisation transforms the problem geometry into a mesh of discrete elements that form an approximation of the geometry. A mesh can be built up by elements from a broad set of element categories where their applicability depends on the nature of the problem and how the problem should be approximated.

There has been no extensive treatment of the discretisation process in this work but an initial conceptualisation of mesh-related concepts and relationships has been modelled. The main reason has been to show how a query language, such as AMOSQL, can be used to model this type of data and how queries can be used to retrieve valuable information, to point out some relevant issues worth mentioning.

A basic type taxonomy, illustrated in Figure 35, has been designed that has a similar structure to the one designed for the geometry model illustrated in Figure 28. The structure of a finite element can here be described in terms of volume elements, surface elements, curve elements, and nodes. These basic element entities are related by the faces, edges, and nodes relationships shown in Figure 36. These relationships describe the element topology and could be compared to the topological relationships for the geometry in Figure 29. Hence, the basic structure of finite element concepts and geometrical concepts are very similar and the idea is that this similarity should facilitate the representation of relationships between the geometry and the mesh. By providing useful relationships between these two models, it is easier to map other quantities that require representations in both models. For instance, one would like to be able to specify different forms of boundary conditions on the geometry but for the analysis they need to be mapped to discretised representations. These ideas are similar to those found in Finnegan et al. [148].

The example in Figure 31 will be continued in this section to introduce these discretisation concepts. This geometry model can be used as the basis for the specification and generation of a finite element mesh. If a simple mesh is specified, with three bi-linear elements per edge of the rectangular, the resulting mesh will have the appearance shown in Figure 37.

As in the case of the geometry model, the type taxonomy is implemented as an AMOSQL types as:

```
create type fea_object subtype of named_object;

   create type element subtype of fea_object;
      create type volume_element subtype of element;
      create type surface_element subtype of element;
      create type curve_element subtype of element;

   create type node subtype of fea_object;
   create type load subtype of fea_object;
      create type line_load subtype of load;
      create type point_load subtype of load;

   create type displacements subtype of fea_object;
      create type fixed_displacements subtype of displacements;
```

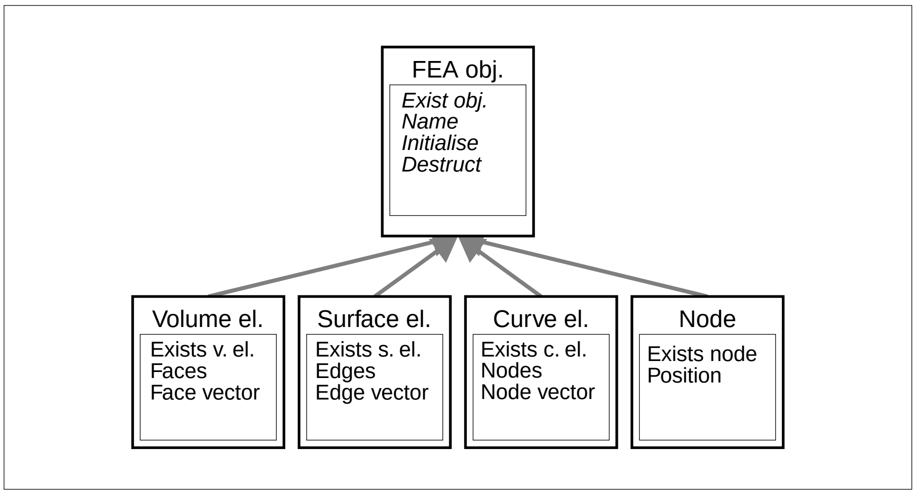


Figure 35. *Basic type taxonomy for finite element-related concepts in the application domain.*

A few extra types that should be used to model loadings and displacements conditions have also been added in this example.

Likewise, the topological element relations are implemented as AMOSQL functions in a similar way to the geometry case.

```
create function face_vector(volume_element ve) -> vector
                    as stored;

create function faces(volume_element ve) -> surface_element as
                    select element(face_vector(ve));

create function edge_vector(surface_element se) -> vector
                    as stored;

create function edges(surface_element se) -> vector
                    select element(edge_vector(se));
```

```
create function edge(surface_element se) -> curve_element as
                    select element(edge_vector(se));

create function node_vector(element e) -> vector
                    as stored;

create function nodes(element e) -> node as
                    select element(node_vector(e));
```

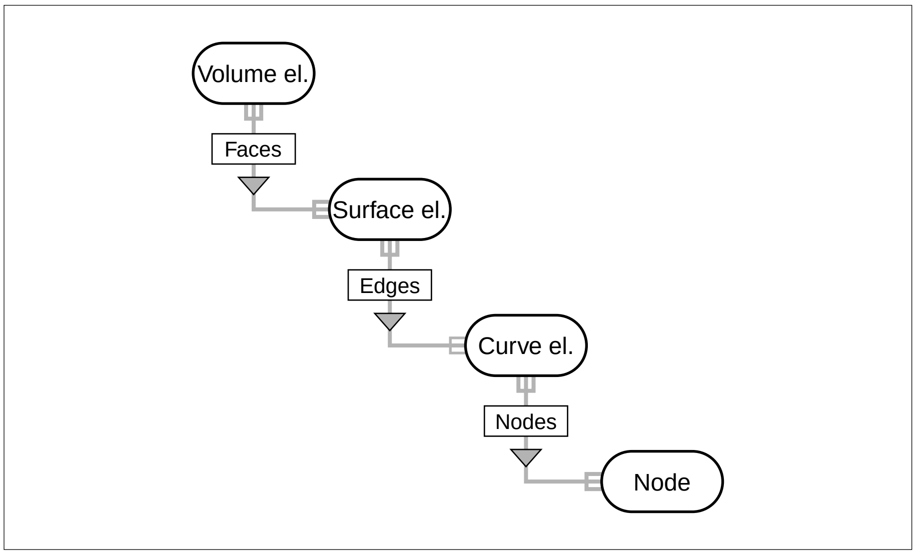


Figure 36. *Topological relationships between basic FEA element concepts.*

It is then possible to populate a database with element and node objects. This is done by continuing to populate our geometrical database. Thus, the element structure in Figure 37 can be created by the following AMOSQL statements:

```
create node (name) :n1 (“n1”),
                   :n2 (“n2”),
                   ...
                   :n16 (“n16”);

create curve_element (name, node_vector)
          :ce1 (“ce1”, {:n1,:n2}),
          :ce2 (“ce2”, {:n2,:n3}),
          ...
          :ce24 (“ce24”, {:n15,:n16});
```

```
create surface_element (name, edge_vector)
          :se1 (“se1”, {:ce1,:ce5,:ce8,:ce4}),
          :se2 (“se2”, {:ce2,:ce6,:ce9,:ce5}),
          ...
          :se9 (“se9”, {:ce17,:ce21,:ce24,:ce20});

create volume_element (name, face_vector)
          :ve1 (“ve1”, {:se1}),
          :ve2 (“ve2”, {:se2}),
          ...
          :ve9 (“ve9”, {:se9});
```

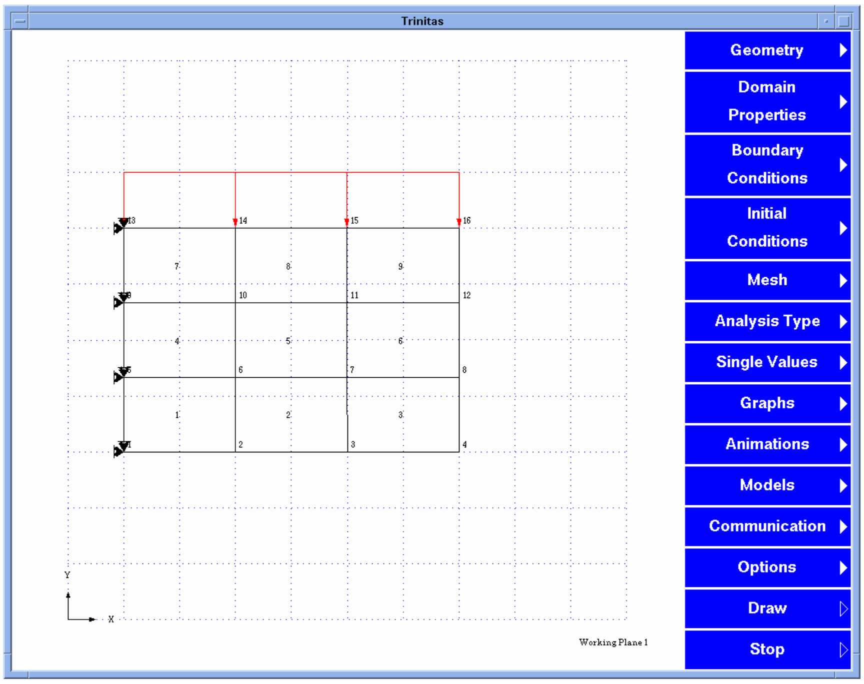


Figure 37. *A simple FE mesh consisting of 9 bi-linear elements including node and element numbers. Rigid boundary conditions are introduced for the left edge and the loading condition is modelled by a line load. Note that node and element numbers are included only for facilitating interpretation of the examples and is not required (but optional) by the FEAMOS system.*

The node numbers and the surface and volume element numbers correspond to the numbering in the mesh in Figure 37, whereas the numbering of the curve elements is given in Figure 38. To facilitate the interpretation of the element and node structure an instance model for one of the nine elements is given Figure 39. There are actually nine similar instance models that share elements and nodes on several levels.

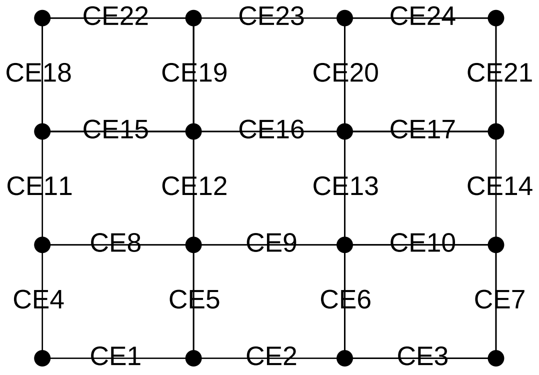


Figure 38. *The labels assigned to the curve elements in the mesh example illustrated in Figure 37.*

It is now possible to state queries to this database model of the mesh. For instance, the nodes of a curve element can be retrieved:

```
name(nodes(:ce2));
“n2” “n3”
```

Another example retrieves the edges of a surface element:

```
name(edges(:se2));
“ce2” “ce6” “ce9” “ce5”
```

These basic functions can further be overloaded, as derived functions, on additional types to get a convenient retrieval capability. For instance, the `nodes` and the `edges` functions are here overloaded on additional element types.

```
create function nodes(surface_element se) -> node n as
                 select unique(nodes(edges(se)));

create function nodes(volume_element ve) -> node n as
                 select unique(nodes(edges(faces(ve))));
```

```
create function edges(volume_element ve) -> curve_element ce as
                 select edges(faces(ve));
```

With this capability, we can easily retrieve nodes for surface elements and nodes and edges for volume elements.

```
name(nodes(:se2));
“n2” “n3” “n7” “n6”

name(nodes(:ve1));
“n1” “n2” “n6” “n5”

name(edges(:ve1));
“ce1” “ce5” “ce8” “ce4”
```

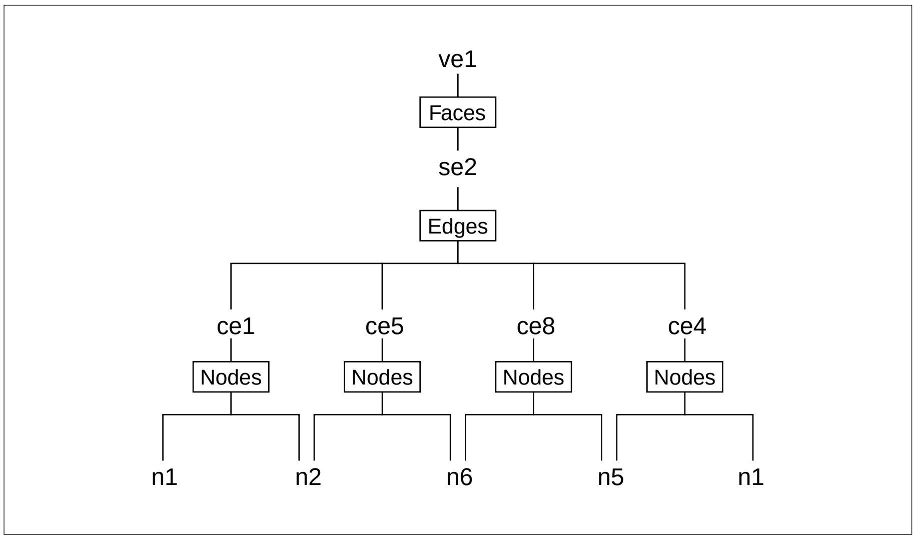


Figure 39. *The structure of an instance model for one of the volume elements in the mesh example.*

Furthermore, the same functions can be applied in the reverse direction to find out in which elements a specific node appears.

```
select name(ve) for each volume_element ve where nodes(ve) = :n1;
“ve1”

select name(ve) for each volume_element ve where nodes(ve) = :n10;
“ve4” “ve5” “ve7” “ve8”
```

So far, this description has only concerned the mesh structure itself, but it is now time show how it can be connected to the geometry. A good idea is to separate the discretisation and the geometry as much as possible, which has already accomplished. Further, the discretisation is probably more dependent on the geometry than the opposite. By defining functions between mesh and geometry on the mesh concepts, the geometry conceptualisation becomes independent of the discretisation conceptualisation. Hence, mesh concepts are tied to the corresponding geometric concepts by the following functions.

```
create function volume(volume_element ve) -> volume as stored;
create function surface(surface_element se) -> surface as stored;
create function curve(curve_element ce) -> curve as stored;
create function point(node n) -> point as stored;
```

These functions must further be populated by appropriate values.

```
set volume(:ve1) = :v1;
set volume(:ve2) = :v1;
...
set volume(:ve9) = :v1;

set surface(:se1) = :s1;
set surface(:se2) = :s1;
...
set surface(:se9) = :s1;

set curve(:ce1) = :c1;
set curve(:ce2) = :c1;
...
set curve(:ce18) = :c4;

set point(:n1) = :p1;
set point(:n4) = :p2;
set point(:n16) = :p3;
set point(:n13) = :p4;
```

Now, relationships across are established and information can be retrieved about these relationships. For instance, it is possible to retrieve the position of the points that define the curve that the curve element :ce11 is associated to.

```
select position(p) for each point p, curve c
           where curve(:ce11) = c and
                 vertices(c) = p;
<10.,70.,0.>
<10.,30.,0.>
```

Maybe it would be more interesting to be able to extract information about loading and displacement conditions. To show how this can be accomplished a few properties are defined for line loads and fixed displacements. To be able to define where the load or the fixation should act, a curve function is defined for each concept. For the line load an additional intensity function is defined to represent the load intensity.

```
create function curve(line_load l) -> curve as stored;
create function intensity(line_load l) -> real as stored;
create function curve(fixed_displacements d) -> curve as stored;
```

Then, a specific load and fixation is created:

```
create line_load(name, curve, intensity)
                    :ll1(“ll1”, :c3, -1000.0);

create fixed_displacements(name, curve) instances
                    :fd1(“fd1”, :c4);
```

When the discretised model should be established in an FEA, one would like to know which nodes boundary conditions are acting on. These nodes can be retrieved in our model. For example, the nodes that are influenced by the line load can be retrieved by the following query:

```
select distinct[1] name(n) for each node n, curve c,
                 curve_element ce, line_load l
              where curve(l) = c and
                    curve(ce) = c and
                    nodes(ce) = n;
“n13” “n14” “n15” “n16”
```

Similarly, the nodes that are influenced by the fixed line is found through the query:

```
select distinct name(n) for each node n, curve c,
              curve_element ce, fixed_displacements fd
                 where curve(fd) = c and
                       curve(ce) = c and
                       nodes(ce) = n;
“n1” “n5” “n9” “n13”
```

Furthermore, if would be possible to eliminate the node that takes part in both the load and the fixation:

```
select distinct name(n) for each node n, curve c,
                    curve_element ce, line_load l
                 where curve(l) = c and
                       curve(ce) = c and
                       nodes(ce) = n and
                       notany(fixed(n));
“n14” “n15” “n16”
```

1. The `distinct` keyword tells the select statement to eliminate duplicates in a similar way to the `unique` operator.

In the preceding query a function, `fixed`, is used that checks if a specific node is fixed and has the following implementation:

```
create function fixed(node n) -> node as
   select n where n = (select distinct m for each node m,
                  curve c,
                  curve_element ce,
                  fixed_displacements fd
             where curve(fd) = c and
                  curve(ce) = c and
                  nodes(ce) = m);
```

Hence, it has been shown how the query language can be used for extracting information about the mesh and its relationships to geometry and boundary conditions. Several of these query examples are much more complicated to express in an ordinary programming language. However, the conceptualisation must be further developed to incorporate and handle the different element types provided in TRINITAS as outlined in Figure 40.

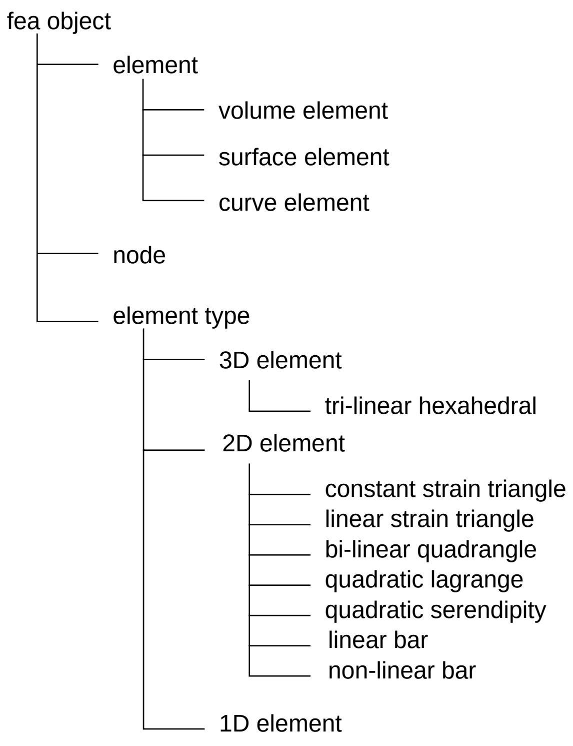


Figure 40. *Outline of a type taxonomy for finite element mesh concepts.*

Furthermore, it must be evaluated whether the data representations currently provided in AMOS are efficient enough to support discretisation operations. If more efficient data representations are required, a similar approach to the matrix data source, described in Section 5.2.3, might be applicable where foreign and specialised data representations were implemented and made available from AMOS. For this purpose, it might be suitable to apply a tree data structure as described in Fenves [39].

Finally, it would be most interesting to investigate if function materialisation techniques [149] could be applied in this context to automatically handle computation and caching of, for instance, derived topology functions.

5.3.3 Finite element analysis solution algorithms

The ability of the DBMS to solve linear equation systems in the FEA process was one primary intention of the design and implementation of the linear algebraic matrix algebra package that were described in Section 5.2.

For this reason, the type system of AMOS was extended with a set of matrix types equipped with a basic set of matrix operations that were transparently made available in the query language. This ability permitted application-specific domain concepts to be given a suitable representation that were transparently handled through the query language. Hence, effective data representations could be provided by the DBMS that eliminate or minimise low-level data administration such as physically copying and transforming data.

The solution algorithms of finite element analysis include procedures for numerically solving linear equation systems. In the FEAMOS system, these algorithms can be expressed in a very natural and high-level terminology by the built-in package for linear matrix algebra. Application-specific domain concepts that are to be represented by matrices are modelled as subtypes of appropriate matrix types. This makes it possible to express the corresponding domain operations in terms of these domain-specific matrix concepts.

We exemplify this by the solution of the previously stated, Eq. (30), linear equation system **K a** = **f**, where **K** is the stiffness matrix, **a** is the displacement vector, and **f** is the load vector. Using domain-specific types, this solution process can be stated in the following manner. According to Figure 41, the stiffness matrix, **K**, is modelled as a subtype of the symmetric_matrix type; the displacement vector, **a**, and the load vector, **f**, are modelled as subtypes of the column_matrix type. Further, by providing a specific domain type taxonomy for these domain concepts, domain-specific properties can be represented independently of the mathematical matrix concepts.

However, the corresponding domain operations must be able to generate the correct result type in order to maintain consistency. In the present case, the solution of the linear equation system is performed by applying the operations that correspond to Eqs. (67), in Section 5.2.3. We see that:

$$\mathbf{K}^b = (\mathbf{U}^T)^f \cdot \mathbf{D}^f \cdot \mathbf{U}^f \tag{69}$$

the factorisation of **K** should return a diagonal matrix and at least one upper unit triangular matrix. If we, for instance, consult Hughes [23], we see that the LDL factorisation produces a diagonal matrix, **D**, where the matrix elements have the same unit as the elements of **K**. Further, for the upper and lower unit triangular matrices, **U** and $\mathbf{U}^T$, the elements are unitless. Continuing with the solution of the upper triangular equation system,

$$\mathbf{U}^b \cdot \mathbf{a}^f = \mathbf{x}^b \tag{70}$$

this operation should produce a displacement vector **a**. When considering the *bbf* binding pattern, this operation should produce a displacement vector **x**, since **U** is unitless. Consistently, the next diagonal scaling operation

$$\mathbf{D}^b \cdot \mathbf{x}^f = \mathbf{y}^b \tag{71}$$

should generate a displacement vector **x** with the *bfb* binding pattern, whereas the *bbf* binding pattern should return a load vector **y**, due to the multiplication of stiffness elements in **D** with displacement elements in **x**. Finally, for the operation that solves the lower triangular system

$$(\mathbf{U}^T)^b \cdot \mathbf{y}^f = \mathbf{f}^b \tag{72}$$

a load vector **y** should be produced since **f** is a load vector and **U** is unitless. Consequently, the *bbf* binding pattern for this operation returns a load vector with the present arguments.

If the original matrix multiplication operations have been carefully designed, they will produce the correct result type for several cases. For instance, in Eqs. (70) and (72), the correct type will be produced since it is deduced from one of the argument or result types. However, when the result type is dependent on combinations of argument and result types, as in Eq. (71), special treatment is needed. In the present case, this function is overloaded for the domain types signatures. Likewise, the result type of **D** in the factorisation operation, corresponding to Eq. (69), is domain-dependent. Here, this is accomplished by overloading this operation on **K**.

By using these basic matrix operations, the solution of the system can be stated in AMOSQL in a very convenient notation. Presuming that a stiffness matrix and a load vector exist and are bound to the interface variables :k and :f respectively, the

AMOSQL query for solving the displacements would look like:

```
select a for each displacement_vector a where :k * a = :f;
```

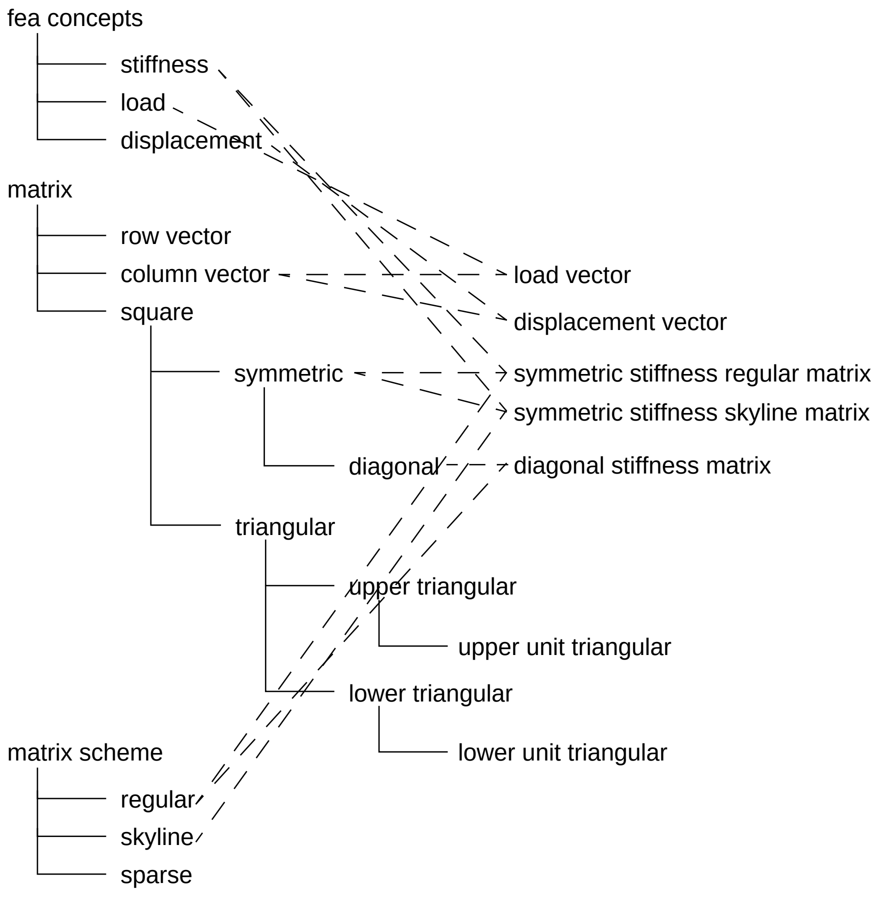


Figure 41. *The representation of application domain types, such as stiffness, loads, and displacements, as subtypes of matrices in the matrix taxonomy. The taxonomy is here somewhat reduced to simplify the presentation.*

The stiffness matrix, **K**, could have been modelled with either of the available representations, i.e. as a subtype of the `regular` or `skyline` matrix schemes that implies a regular or a skyline representation of the equation system. However, this does not change the expression for the solution algorithm and the system automatically creates matrices with the appropriate types and selects the correct operations to solve the equation system. By providing several matrix representations schemes, the application can explicitly choose a suitable representation depending on the problem type, or implicitly, by letting the query processor make the decision based on type information or a potential cost model. The DBMS could also support the derivation of the correct type representation by using a derived type mechanism as described in Werner [93], but this facility was not available at the time of this work. Thus, the potential convenience of using derived types must be further evaluated.

If the solution algorithm were embedded within an application, it would be suitable to represent it as a database function that could be accessed from the application. This is expressed as follows:

```
create function lin_solve(symmetric_stiffness_regular_matrix k,
                        load_vector f) -> displacement_vector a as
                    select a where k * a = f;
```

and overloaded for the skyline representation:

```
create function lin_solve(symmetric_stiffness_skyline_matrix k,
                        load_vector f) -> displacement_vector a as
                    select a where k * a = f;
```

A small example[1] in Figure 42 is used to show how this can be used in problem solving.

The compact representation using skyline matrices is assumed in this example. The creation of the basic matrices and the solution of the corresponding equation system are performed by the following AMOSQL statements:

```
set :k = construct(“symmetric_stiffness_skyline_matrix”,{{1.0},
                  {-1.0,2.0},{-1.0,2.0},{-1.0,2.0}}});

set :i = construct(‘iarray’,{0,2,4,6});[2]

set dindex(:k) = :i;

set :f1 = construct(“load_vector”,{1.0,0.0,0.0,0.0});

set :a = solve(:k,:f1);

print(:a);
{4,3,2,1}
```

1. Normally, matrices are created and updated by calling database functions from the application which are inconvenient to use in textual examples. The syntax for interactive manipulation of matrices is somewhat temporary and has been slightly modified to shorten the presentation of the examples.
2. It is not necessary to create the index array :i of the skyline matrix :k explicitly but this is the current procedure.

It is further possible to express the repetitive solution of the same problem for different load cases by overloading the solve function on bags[1] of load vectors. This would be expressed as:

```
create function lin_solve(symmetric_stiffness_skyline_matrix k,
                        bag of load_vector f) ->
                        displacement_vector a as
                 select a where k * a = element(f);
```

$$\begin{bmatrix} 1 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix} \cdot \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 2 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Figure 42. *An example equation system with a 4x4 stiffness matrix, a displacement vector with 4 unknowns, and two load vectors including one non-zero component each.*

Combining the load_vector `:f1` with another called `:f2` into a bag `:fseq` makes it possible to express the corresponding solution in the following manner:

```
set :f1 = construct("load_vector",{1.0,0.0,0.0,0.0});

set :fseq = {:f1,:f2};

set :aseq = lin_solve(:k,:fseq);

print(:aseq,0);
{4,3,2,1}

print(:aseq,1);
{8,6,4,2}
```

It should be noted that, by overloading matrix operations, the domain conceptualisation will hold independently of how each concept is represented or how the operations are implemented. As discussed earlier, the declarativeness of the query language makes it possible to achieve both logical and physical data independence. Since the schema in

1. Actually, one would like to have a `sequence of load_vector` instead of a bag (or multiset) to preserve order but, as have been pointed out earlier, this capability is not currently available in AMOSQL.

our case not only includes data descriptions, but also operator descriptions, this independence of physical and logical implementation becomes valid for operations too.

For instance, in the present context the solution of a triangular equation system **A b** = **c**, where **A** is a triangular matrix and **b** and **c** are column vectors, is expressed in AMOSQL as:

```
select b for each column_vector b where :A * b= :c;
```

This expression is independent of how the C-procedure that solves this system is implemented which results in physical implementation independency.

Furthermore, by replacing the name of the variables in this expression, the expression solves a linear equation system **K a** = **f**, where **K** is a symmetric matrix and **a** and **f** are column vectors. However, this solution includes a transformation of the equation system into a set of simpler equations, according to Eq. (68), that is expressed by an AMOSQL expression. The transformation in Eq. (68) could be replaced by a different transformation that also solves the linear equation systems, for example by applying a different matrix decomposition method. This does not influence the expression of the solution in terms of the times operator on the highest level. Hence, a form of logical implementation independency is achieved.

In this sense, the term “data independence” might be unsuitable to use in some situations and a more suitable term could, for instance, be physical and logical implementation independence.

Since both structure and process of the application domain can be captured, we can model complete parts of the domain knowledge independently of any application. Hence, this provides sharing of domain knowledge among applications and further facilitates its reuse. Furthermore, the ability of the query optimizer to handle domain models including concepts, data representations, operators, and cost models that are domain-dependent, can be viewed as a form of *domain compilation*. Hence, we have a domain compiler that uses domain knowledge in the compilation process.

Further work in this area would be to develop corresponding domain models for other parts of the FEA process, such as calculating element quantities including the element stiffness in Eq. (32) and the element loads in Eq. (33). The possibility of using matrix algebraic expressions for this purpose must be studied in more detail.

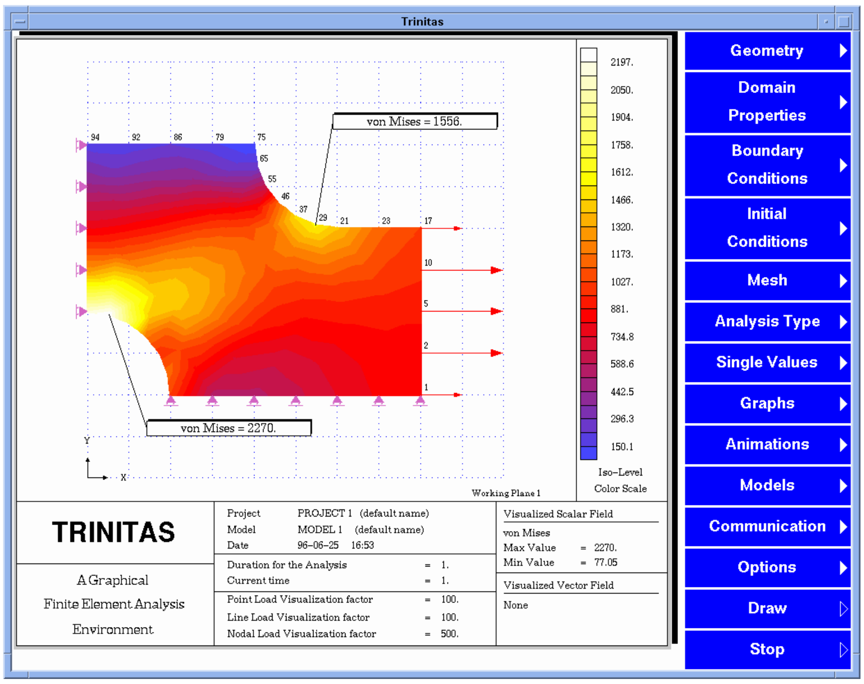
5.3.4 Result evaluation

The result evaluation activity includes calculation and interpretation of various physical quantities, such as stresses, displacements, and eigenvalues that is evaluated according to some design criteria. As illustrated in Figure 43, TRINITAS supports this evaluation graphically by facilities for displaying, for instance, stress and displacement fields, single values, multiple values in graphs. The calculated results are stored in arrays that are accessed by a set of built-in FORTRAN subroutines. By making various menu-based selections, the user can activate appropriate routines and display the result.

When the basic array-based data storage of TRINITAS was replaced by a corresponding array representation in AMOS, it immediately made all data accessible at the array level from the query language. Without any further conceptual modelling, it was possible to state queries over all data, including calculated analysis results.

Since all data are available at the query language level, the user has numerous capabilities to compose queries for retrieving, combining, and transforming data. For instance, queries can be stated for retrieving:

- max and min displacements,
- max and min stresses,
- various weighted quantities such as von Mises stresses, etc., and
- quantities combined with additional constraints.



TRINITAS

A Graphical
Finite Element Analysis
Environment

```
Project     PROJECT 1   (default name)
Model       MODEL 1     (default name)
Date        96–06–25    16:53

Duration for the Analysis               =   1.
Current time                            =   1.

Point Load Visualization factor         =   100.
Line Load Visualization factor          =   100.
Nodal Load Visualization factor         =   500.
```

```
Visualized Scalar Field
von Mises
Max Value    =  2270.
Min Value    =  77.05

Visualized Vector Field
None
```

Figure 43. *An example that shows how analysis results can be presented to the user in TRINITAS. The picture includes coloured iso-levels of von Mises stresses, the displacement field, and two single stress values.*

The following query selects a single value from an array[1] of von Mises stresses, where the index corresponds to a certain node:

```
ref(array_named(“VON1    “),0);
844.779
```

If one would like to extract the maximum von Mises stress this query can be extended to traverse the complete array and then apply the `max` aggregation operator:

```
select max((select vonmises
   for each integer i, integer j, real vonmises
      where   i = iota(0,j) and
         j = size(array_named(“VON1    “)) - 1
         and vonmises = ref(array_named(“VON1    “),i)));
2269.62
```

Further, it is not necessary to store every type of result in the database. Instead, derived functions can be defined that calculate these measures. The ability to do calculations within queries is exemplified in the following ad-hoc query where the previous query is reformulated in terms of the basic stress components:

```
select max((select vonmises
   for each integer i, integer j,
       real sigmaxx, real sigmayy, real sigmaxy, real vonmises
      where j = size(array_named(“VON1    “)) - 1 and
            i = iota(0,j) and
            sigmaxx = ref(array_named(“SXX1    “),i) and
            sigmayy = ref(array_named(“SYY1    “),i) and
            sigmaxy = ref(array_named(“SXY1    “),i) and
            vonmises = sqrt(abs(sigmaxx*sigmaxx +
                                sigmayy*sigmayy -
                                sigmaxx*sigmayy +
                                3*sigmaxy*sigmaxy))));
2269.62
```

Associated information can be retrieved by selecting several results in the same query, as in the subsequent query where the stress value is combined with a number corresponding to the node number:

```
select max((select vonmises,i+1
   for each integer i, integer j, real vonmises
      where   i = iota(0,j) and
         j = size(array_named(“VON1    “)) - 1 and
         vonmises = ref(array_named(“VON1    “),i)));
<2269.62,83>
```

1. The `array_named` function maps a name of an array to an array object.

The query results can further be filtered by adding additional constraints in the query. Here, the von Mises stresses that exceed 2000.0 (MPa) are extracted along with the node numbers.

```
select vonmises,i+1
   for each integer i, integer j, real vonmises
      where   i = iota(0,j) and
         j = size(array_named(“VON1    “)) - 1 and
         vonmises = ref(array_named(“VON1    “),i) and
         vonmises > 2000.0;

<2269.62,83>
<2128.4,90>
```

The previous examples showed how the current array representation was directly accessible from the query language. In comparison to the original TRINITAS program, the expressibility for accessing and composing data have increased dramatically in FEAMOS, even if the queries became a bit clumsy on occasion. In addition, it is not so critical if some relevant evaluation functionality has been overlooked in the implementation. It will be relatively simple to add this through the query language. In TRINITAS, this must be done by implementing some new FORTRAN routines.

As was pointed out, the previous examples implied a rather clumsy and unattractive use of the query language. To get rid of this disadvantage, one could introduce much more structure within the result information through OO modelling. For instance, by associating the stress components to the nodes, the former queries can be expressed and interpreted much more conveniently. This can be accomplished as:

```
create function stresses(node n) -> farray as stored;
```

As we only consider stress components in the xy-plane, the `stresses` function stores an array of three reals that can represent the stress components σ_{xx}, σ_{xy}, and σ_{yy}. These can be explicitly referenced and defined by derived functions.

```
create function sigma-xx(node n) -> real as
                 select ref(stresses,0);

create function sigma-xy(node n) -> real as
                 select ref(stresses,1);

create function sigma-yy(node n) -> real as
                 select ref(stresses,3);

create function vonmises(node n) -> real vm as
                 select sqrt(abs(sigma-xx(n)*sigma-xx(n) +
                                sigma-yy(n)*sigma-yy(n) -
                                sigma-xx(n)*sigma-yy(n) +
                                3*sigma-xy(n)*sigma-xy(n)));
```

By structuring the stresses along these lines, the previous query for retrieving the maximum von Mises stress can now be expressed as:

```
select max((select vonmises(n) for each node n));
```

Likewise, the node[1] can be retrieved together with the von Mises stress:

```
select max((select vonmises(n),n for each node n));
```

Additional constraints can easily be introduced in a similar way to the former example, but here the there is a more direct association to the stress function.

```
select vm for each node n, real vm
          where vm = vonmises(n) and vm > 2000.0;
```

It will also be easy to express queries including additional constraints as in the following query where von Mises stresses within a certain area are extracted:

```
select n for each node n where vonmises(n) > 0.0 and
                              x_coordinate(n) > 50.0 and
                              x_coordinate(n) < 70.0 and
                              x_coordinate(n) > 40.0 and
                              y_coordinate(n) < 60.0;
```

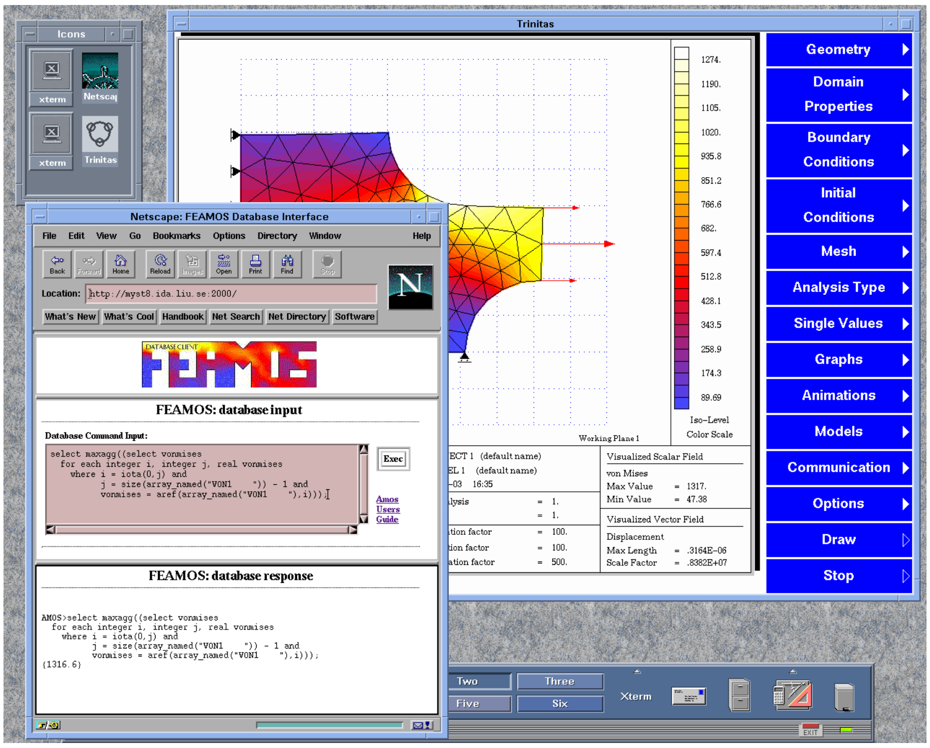
Here, it is presupposed that coordinate functions are defined for nodes.

By overloading stress functions on other types, in addition to nodes, it is possible to select stresses along boundary, curves, or other geometrical sections.

Currently, we have the possibility to directly interact with the FEAMOS database through the query language through a database client, shown in Figure 44. This www-based database client uses a general www-interface implemented in AMOS. Within this work we have also discussed and planned to implement functions for displaying the result of a query graphically on the analysis model. This type of display function should, for instance, be able to display a set of nodes that have been filtered out as a result of a query.

Within this context there are also two database mechanisms that ought to be studied in further research. As indicated in other areas of FEA, indexes can be of great importance for supporting efficient processing. FEA results, such as stress and temperature fields, are examples of spatial quantities where spatial indexing techniques could be applied. Furthermore, in studying the issue of whether a stored or a derived representation of analysis results is preferable, it should be of interest to evaluate how *function materialisation techniques* [149] could support these decisions in an automatic manner.

1. Note that the node is here an object and not a number as in the previous example.



```
select maxagg((select vonmises
  for each integer i, integer j, real vonmises
    where i = iota(0,j) and
          j = size(array_named("VON1    ")) - 1 and
          vonmises = aref(array_named("VON1    "),i)));
```

```
AMOS>select maxagg((select vonmises
  for each integer i, integer j, real vonmises
    where i = iota(0,j) and
          j = size(array_named("VON1    ")) - 1 and
          vonmises = aref(array_named("VON1    "),i)));
{1316.6}
```

Figure 44. *An illustration of the use of a www-based database client in combination with FEAMOS for providing ad-hoc query capabilities.*

Even if the user can get great support from graphics and a query language, as described here, the result interpretation is mainly a manual activity. The efficiency and quality could probably be increased for this activity by taking advantage of the rule system in AMOS. By defining active rules that interpret the results, parts of the evaluation activity could probably be automated. The applicability of these ideas requires additional study.

5.4 PERFORMANCE ISSUES

As pointed out earlier in this thesis, performance is of vital importance for FEA applications since large data volumes, and complex and costly operations are involved. It is, moreover, important to attain good performance on the average as well as good scale-up characteristics. Furthermore, for interactive FEA applications it is important that the response time is acceptable for the activities that are to be performed.

Since this work does not cover a final and complete implementation of an FEA application, it is not possible to make a complete performance evaluation either. However, the FEAMOS system is complete in the sense that it is possible to perform complete analyses to demonstrate its current capabilities. This makes it possible compare the performance of subactivities in the analysis process where alternative implementation techniques exist for FEAMOS with respect to TRINITAS.

One of the first performance evaluations made in this work, [7], was to compare the efficiency of FEAMOS and TRINITAS of creating and accessing simple data entities. A major reason was to evaluate the interface overhead for accessing the database in FEAMOS compared to accessing a Fortran array in TRINITAS. The performance test measured the time for the creation of objects including an initialisation (update) of two stored functions. This corresponds to point objects, a name attribute, and a position attribute with three coordinates. We also measured the access time for random access of the same number of objects including the access of the name and the position function. These results are presented in Figure 45 and Figure 46. The original figures in Orsborn [7], were measured with WS-IRiS, [118], and these have later been updated to be valid for AMOS.

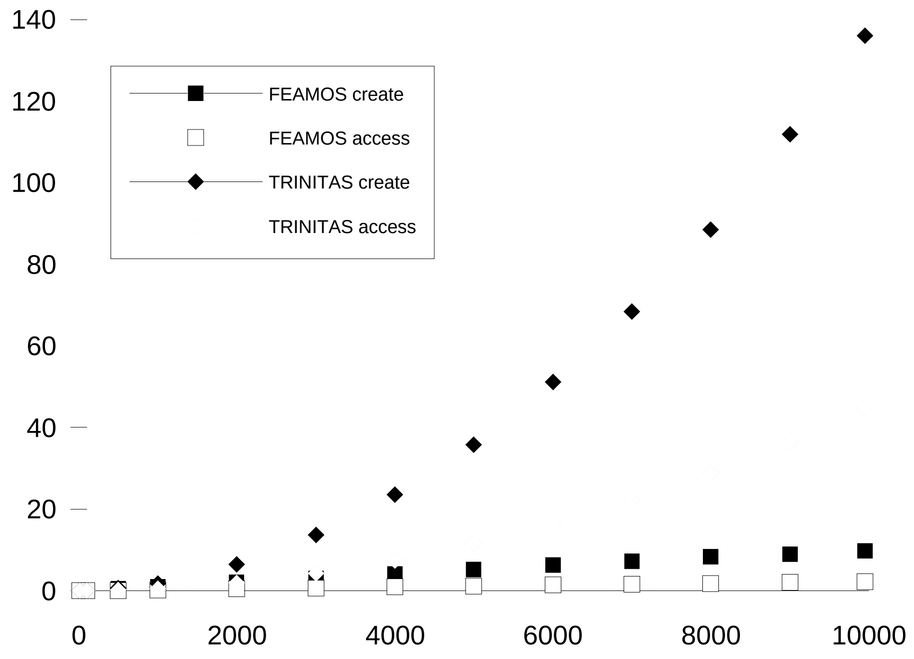


Figure 45. *A performance comparison between FEAMOS and TRINITAS. The diagram shows the real time for creating point objects and for accessing the position of randomly chosen point objects. The x-axis represents the number of objects and the y-axis the real execution time in seconds.*

It is seen from the diagrams that TRINITAS has quadratic performance curves while the FEAMOS curves show a linear behaviour. The improved behaviour is explained by the lack of dedicated storage structures in TRINITAS and results in a significantly improved performance of FEAMOS in comparison to TRINITAS when the amount of data increases. TRINITAS performs linear search over the point object set, whereas FEAMOS takes advantage of built-in storage structures of AMOS, such as hash tables, for efficient access. These kinds of storage structures can, of course, also be implemented in TRINITAS, which should result in a similar inclination of the TRINITAS performance curves. However, Figure 45 illustrates the importance of efficient indexing techniques to provide good scaling capabilities.

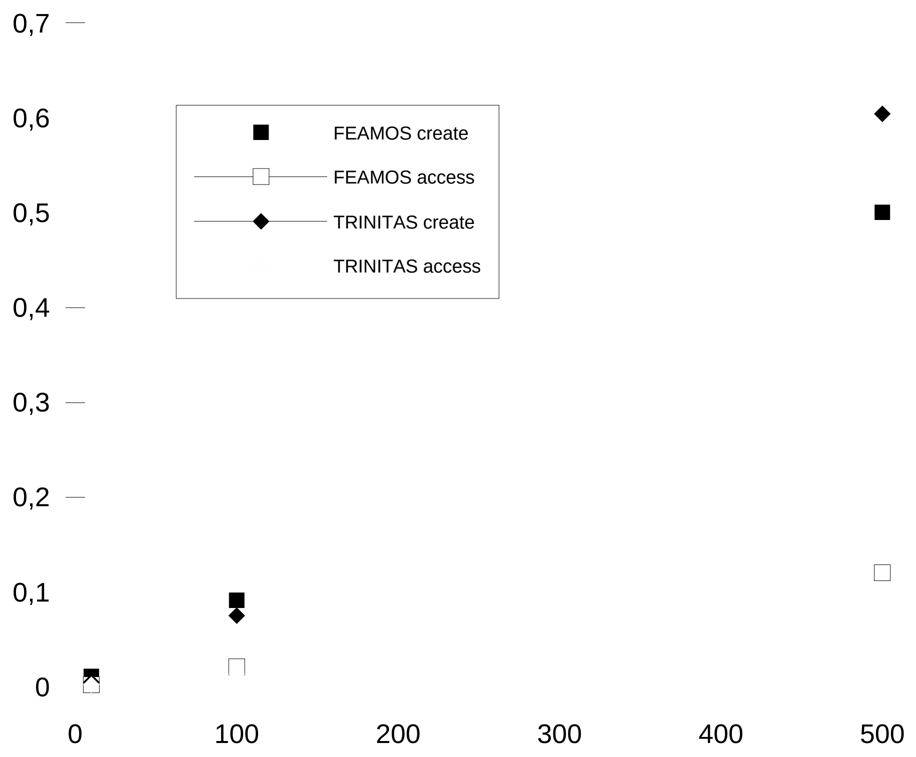


Figure 46. *The diagram shows a more detailed view of the previous diagram for creating and accessing point objects that reveals the performance of FEAMOS and TRINITAS for small data sets. The x-axis represents the number of objects and the y-axis the real execution time in seconds.*

As seen in Figure 46, TRINITAS performs better than FEAMOS at small sets of point objects, i.e smaller than about 150. This is due to the interface overhead in accessing the database. Nevertheless, it shows that FEAMOS performs fairly well even with small data sets[1] and the processing performance is no real bottle-neck at these object volumes.

Here it is worth mentioning that a constant access overhead can significantly degrade performance if the access frequency of application-specific operations is high, for example, if a large number of small data items are accessed one at a time instead of as a whole. Another example would be when efficient indexing techniques are applied in the database but the data filtering is performed in the application algorithms without indexing. Hence, efficient representation methods, like indexing techniques, must be applied intelligently and accompanied by appropriate processing methods and algorithms within the application or the database. If an existing application is to take full advantage of database facilities, such as indexing, this might require that certain application operations must be redesigned. Concerning, the comparison in Figure 45 and Figure 46, it should further be noted that the access phase is more critical than the creation phase since it has a higher frequency in a real application situation.

The ability to extend the DBMS with tuned and efficient data representations has been emphasised here as an important capability for computational database technology. Therefore, it was of interest to evaluate the performance of our array implementation. This has been done by measuring the execution time for decomposing the stiffness matrix. This operation concerns the stiffness matrix represented as a single array which is operated upon by the decomposition method implemented in Fortran (the same base routine is used in both FEAMOS and TRINITAS in this comparison).

The outcome of this comparison is graphically presented in Figure 47 and the measured values are provided in Table 1. These show that the FEAMOS array implementation is at least as fast as the corresponding Fortran array in TRINITAS.

The last performance test was made with the intention to test whether it was possible gain an acceptable performance for real and critical FEA analysis activities. The establishment and solution of the complete linear equation system was chosen as a relevant test case. No tuning has been made of the interface routines that are involved and since database access is frequent when establishing the stiffness matrix and load vector, it was expected that FEAMOS should perform significantly worse than TRINITAS. The results from these measures are presented in Figure 48 and Table 2, and in Figure 49 and Table 3.

These figures show that in the worst case FEAMOS is about five times slower than TRINITAS and, at best, FEAMOS is about 15 percent behind. This is significantly better than was expected considering the fact that the database access in FEAMOS include some Lisp interpretation as well. Hence, even if these figures do not show any ultimate

1. It should be noted that no extensive exploration of the tuning possibilities of the interface has been carried out and it is expected that this could further reduce the interface overhead.

FEAMOS performance, they indicate that the FEAMOS approach is a promising direction for continued work.

In comparing Figure 48 and Figure 49, it is seen that there is a bigger difference between FEAMOS and TRINITAS in the first diagram than in the second. This is probably due to the fact that the rate between the number of numerical operations and number of data accesses is higher for problems including quadratic elements then those including linear elements.

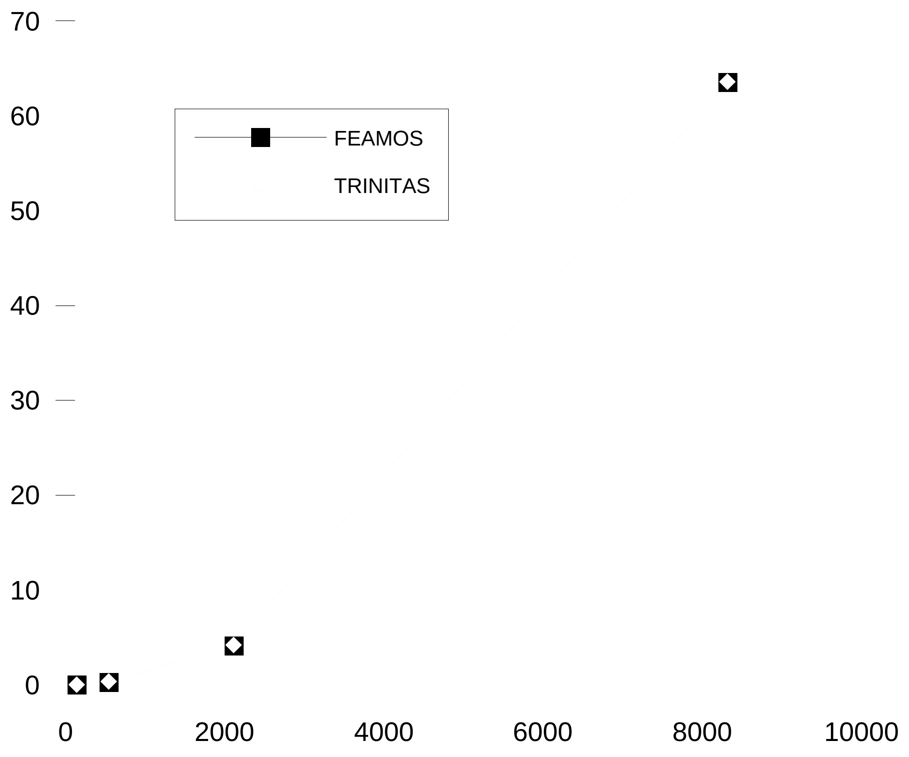


Figure 47. *A diagram showing a comparison of the real execution time in FEAMOS and TRINITAS for the decomposition of stiffness matrices of different sizes, reflecting the relative efficiency of the array representation in FEAMOS. The x-axis represents the number of unknowns and the y-axis the real execution time in seconds.*

The present work has not included any detailed analysis of storage requirements in FEAMOS compared with the corresponding requirements of TRINITAS. Furthermore, this matter raises additional questions that might need an answer, for instance, which parts of data should be stored and what parts can be derived with respect to storage and processing requirements. These types of decisions would preferably be delegated to the DBMS. These issues have mainly been left to future research.

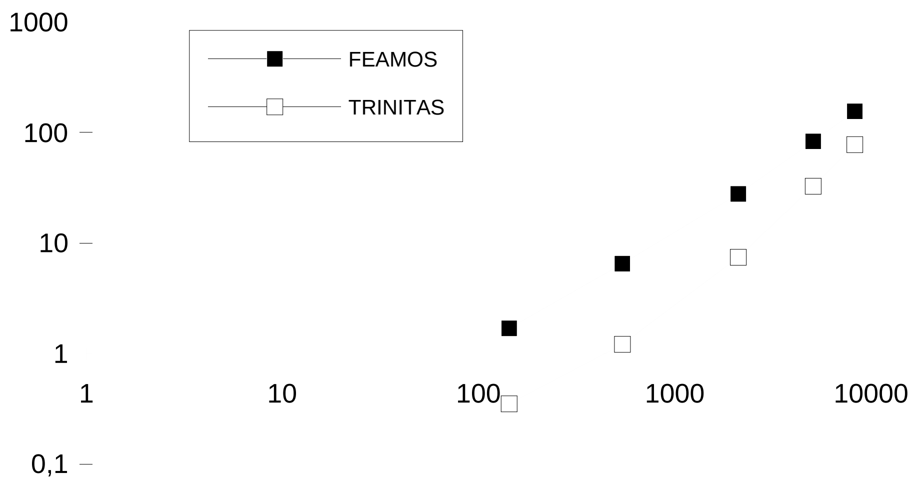


Figure 48. *These two graphs show the real execution time for establishing and solving complete equation systems for models of linear elements. The x-axis represents the number of unknowns and the y-axis the real execution time in seconds.*

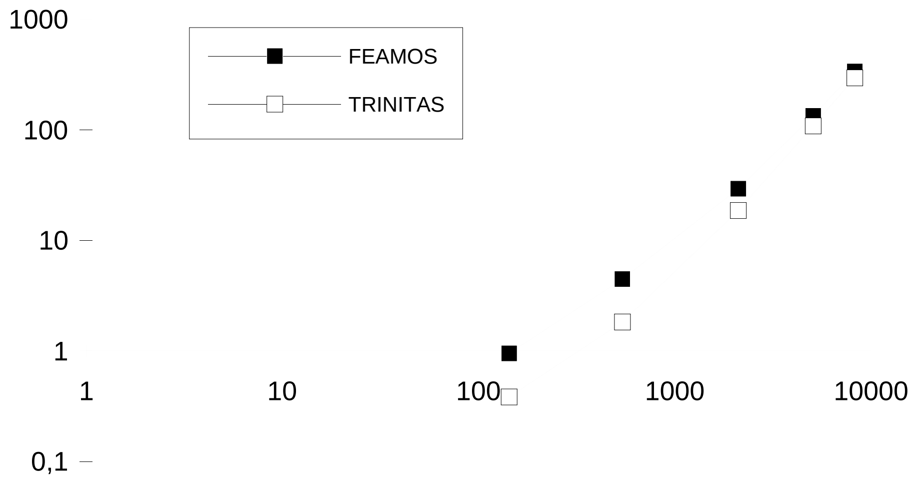


Figure 49. *The same comparison as in the previous diagram except that the models include quadratic elements. The x-axis represents the number of unknowns and the y-axis the real execution time in seconds.*

To traditionalists it might be surprising that the performance results showed that the FEAMOS system, including a DBMS, could perform this well. Major contributing factors to these encouraging performance measures of FEAMOS also include the fact that the database is embedded (shares the same address space) in the application and that the database is in main-memory. Disk access and process or network communication can severely slow down the system.

Problem	Degrees of freedom	FEAMOS	TRINITAS
8x8	144	0,02	0,03
16x16	544	0,30	0,31
32x32	2112	4,15	4,17
64x64	8320	63,54	63,59

Table 1. *Numerical values for the comparison of the real execution time in FEAMOS and TRINITAS for decomposing stiffness matrices of different sizes that were graphically presented in Figure 47.*

Problem	Degrees of freedom	FEAMOS	TRINITAS
8x8	144	1,69	0,35
16x16	544	6,57	1,21
32x32	2112	27,84	7,50
50x50	5100	83,08	32,91
64x64	8320	156,98	78,34

Table 2. *Numerical values for the graphical presentation of solving complete equation systems in Figure 48.*

The performance evaluations made so far have compared FEAMOS with TRINITAS, the original FEA program. These comparisons have provided some valuable insight for this work and additional evaluations are required in future work. In further studies, it would also be valuable to make comparisons of both FEAMOS and TRINITAS to commercial FEA software.

Problem	Degrees of freedom	FEAMOS	TRINITAS
4x4	144	0,96	0,38
8x8	544	4,46	1,82
16x16	2112	29,55	18,65
25x25	5100	134,59	109,72
32x32	8320	339,27	298,57

Table 3. *Numerical values for the graphical presentation of solving complete equation systems in Figure 49.*

6 RELATED TECHNOLOGIES

This chapter will review the main alternative implementation technologies that could be considered in this context. As reviewed in Chapter 2, object-oriented programming (OOP), database technology, and knowledge-based technologies, have been applied to implement FEA applications. Within the field of general database technology the main implementation alternatives are the relational or OO database technologies discussed in Chapter 3.

Another important and related field is the emerging standardisation of specification and exchange of product data within the STEP standard [77]. This standard includes parts that cover FEA data as well as other related data, such as CAD data.

6.1 IMPLEMENTATION TECHNOLOGIES

Object-oriented programming has been applied in implementing FEA applications by several researchers as shown in Section 2. OOP provides OO modelling capabilities, such as objects, attributes, methods, inheritance, and encapsulation. Various OO languages support these capabilities to different degrees. In general, OOP supports physical data independence through encapsulation. In comparison to database technology, encapsulation supports a more limited form of physical data independence since the

DBMS also takes advantage of optimization and indexes. Furthermore, the view mechanism of DBMSs supports logical data independence which is not available in OOP. For example, views can take advantage of query optimization whereas a derived attribute in an OOP must be programmed procedurally. The AMOSQL query language extends the OO modelling capabilities with overloaded multi-directional functions that increase the capabilities of encapsulation and reuse. Further OOP provides programming language execution efficiency which is important for the intended types of applications. Suitable data structures for representing numerical data, such as multi-dimensional arrays, are also available even if specialised representations must be implemented. The storage management of OOP languages are at a low level with little support for large data sets. OOP languages do not provide any general DBMS facilities, including persistence, transactions, queries, optimization, and so on.

An R DBMS provides complete DBMS capabilities, including a relationally complete query language, database schemas, storage management, transactions, and persistence. On the other hand, the relational modelling paradigm is less suitable than the OO modelling paradigm. This can result in more complex schemas for the relational model in comparison to object-oriented schemas [76]. Further, R DBMSs do not support suitable data structures for collections of numerical data. In addition, there is little or no support for adding new data structures. Application-specific operations can sometimes be defined as database procedures stored in the database. More complex schemas and the unavailability of numerical data structures and the corresponding operations are contributing factors that can make it hard to accomplish execution efficiency of the same level as programming languages.

OO DBMSs have modelling capabilities at the same level as OOP languages and can provide execution efficiency at the same level as well. In addition OO DBMSs provide persistence and limited query and optimization capabilities. However, OO DBMSs have no support for facilities, such as overloaded multi-directional operations provided in AMOS, i.e. they can not handle overloading on all arguments and operations can not be defined for different binding patterns. Furthermore, as in OOP, there is no support for views that reduces the data independence capabilities. Extensibility is also restricted to user-defined types and operations (methods). Due to the lack of query optimization is absent, there are no facilities available that correspond to an extensible query processor that can optimize expressions involving application-specific operations.

Earlier experiences, [33] [150] [151] [152], have shown that knowledge-based (KB) implementation techniques usually provide a rich set of modelling capabilities, such as OO modelling variants such as frames, logical formulas, and rules, at least in large hybrid KB tools. These powerful and flexible modelling capabilities are also a drawback since there is no uniform standard for expressing application knowledge. This is especially apparent when there is a need for exchanging data with other applications. The impedance mismatch between the data modelling capabilities of the application and the KB tool makes data exchange hard. Furthermore, KB tools normally rely on heuristic search techniques that are less efficient in comparison to query optimization techniques found in DBMSs. Hence, DBMSs have better scalability performance in comparison to KB-tools. However, several mechanisms from knowledge-based systems have been adopted by the database field and adapted to the efficiency requirements of DBMSs [153]. For example, several DBMSs support constraints, rules, and triggers [107].

These were some general comparisons among alternative implementation technologies; however, to be able to make indisputable conclusions in specific situations real implementations must be compared.

6.2 THE STEP STANDARD AND THE EXPRESS LANGUAGE

An emerging international standard aims to deal with the modelling and exchange of product data. The International Standard ISO 10303 “Industrial Automation Systems and Integration - Product Data Representation and Exchange”, ISO [77], that is usually referred to as STEP, has the objective to

> *“provide a neutral mechanism capable of describing product data throughout the life cycle of a product, independent from any particular system”.*

STEP should provide an “Esperanto” for management of product information independent of the discipline involved and where different levels of usage could range from the application-specific to the inter-enterprise level. ISO 10303 is organized in a series of parts, where the series involves: *overview, description methods, implementation methods, conformance testing methodology and framework, integrated generic resources, integrated application resources, application protocols,* and *abstract test suites.* The overview series describes the structure and contents of the standard and the description method’s series includes the description of the formal data description language EXPRESS, described in Schenk and Wilson [154], and also graphical representation notations, e.g. by means of EXPRESS-G. Further, the implementation method’s series intends to specify different implementation techniques for realizing data sharing, including physical file exchange, application programming interfaces, and database implementations. The conformance testing methodology and framework together with the abstract test suites define the requirements and testing procedures to apply to an implementation to judge if it is in conformance with the ISO 10 303 application protocol. The integrated generic resources, integrated application resources, and application protocols include the actual conceptual schemas for product information. For example, there is one part of integrated application resources that specifies the requirements for exchanging FEA information, [78].

STEP usage could be applied at several levels and there is currently a convention with four implementation levels for defining a STEP system based on its type of data sharing:

1. file exchange,
2. working form (structured data in memory),
3. database, and
4. knowledge base.

The first release of STEP includes the specification of the level 1 file format for data exchange. Further, the central ingredients in STEP include:

1. *the EXPRESS language,* used to specify the
2. *conceptual schemas,*
3. *SDAI* – the STEP data access interface, together with
4. the physical file exchange structure.

These ingredients make it possible to specify, represent, and exchange product information in a formalized and standardized way, independent of the system, software or discipline involved. Thus, two different systems must be able to represent the same types of product data they would like to exchange.

The idea of a standard for representing and communicating product information is of great importance for the development and efficiency of enterprise and engineering information management. It simplifies the management and communication of product information in an inter-enterprise or -organisation situation as well as within enterprises. A standard will also make industry less dependent on certain software or hardware vendors and thus stimulate further evolution of product management systems.

It is, however, important that a standard like STEP continuously evolves and has the aim to integrate future requirements of knowledge management. Then if a standard is flexible and extensible it can support and not hinder the development of product information management of an enterprise.

There is currently no support within STEP for communicating product information by means of a query language. This could, however, be included in a later version of STEP where the database implementations of STEP are addressed. A query language might simplify the representation of application protocols which might be seen as views of the product model. The SDAI might also be represented in a query language taking advantage of general mechanisms for data management currently provided by advanced database management systems.

7 SUMMARY

7.1 CONCLUSIONS

This work covers database technology for FEA applications. The potential of OR database technology in this context has been studied and evaluated by implementing FEAMOS, an FEA application integrated with an OR DBMS. The development of the FEAMOS prototype system has meant that about 5000 lines of C code, and 5000 lines of AMOSQL and Lisp code, have been implemented together with some additional Fortran code. At the same time a number of Fortran routines have been eliminated from TRINITAS. Certain areas have been studied in more detail than others and this selection has mainly been based on the convenience from an implementation perspective rather than giving priority to specific issues. In areas where additional database facilities need to be implemented or where the implementation has not reached a mature state, the discussion and conclusions are based on simplified implementations.

We have presented an architecture for an FEA application that combines an existing FEA program with an OR DBMS. The present approach provides positive effects on both the external and the internal level. Externally, the mediator approach can support, for example, data exchange and transformation, data and operator sharing, data distribution, concurrency control among applications and data sources in an EIS system. By embedding an extensible and main-memory resident OR DBMS in the application, immediate access is gained to general database capabilities such as storage management, data modelling, query language, query processing, and transaction processing. Domain-specific data management can be supplied without sacrificing execution efficiency.

More specifically,

- The FEAMOS approach, an original idea of using main-memory resident, extensible, and OR database technology for FEA, has been introduced.
- An architecture for FEAMOS has been designed that tightly integrates the FEA application with the embedded DBMS. Architectural considerations, such as the influence of data representations and processing location on the overall efficiency have been discussed.
- The architecture of a mediator-based global EIS system incorporating the FEAMOS system has been outlined. It has been shown how the architecture of the mediator system can support sharing, exchange, and combination of data and processing among EIS applications and data sources.
- The applicability of the domain model concept that provides the application with domain modelling, compilation, and optimization capabilities of an OR query language has been demonstrated. The benefits of query language modelling and ad hoc queries have been shown for various FEA activities. Additional benefits of using domain models that include easier access through a query language, better data descriptions (such as schemas), and ad hoc query processing have been presented. Hence, it is not necessary to re-implement low-level dedicated data structures such as indexes for each new system. Such a reimplementation not only duplicates implementation efforts but, as our example shows, may prove less efficient than the highly optimized data management provided by an embedded DBMS. Domain models can also form a base for mediation of domain information among applications and data sources in an EIS system. For example, by providing access to other databases, such as relational DBMSs described in [8] and [123], from the domain model it is possible to build models and ad hoc queries that combine data from other databases.
- AMOS has been extended with foreign data sources for numerical matrix algebra and basic array representations. Specific capabilities of AMOSQL to handle overloaded and multi-directional foreign functions provide a convenient mechanism for expressing and implementing domain-specific operations as found in numerical matrix algebra. This technique provides a more powerful capability of abstraction and reuse in comparison to pure object-orientation. The matrix data source takes advantage of these capabilities for implementing and using type-based function dispatching for selecting appropriate matrix operations. An extension of these facilities to include domain-specific optimization techniques, including dynamic query optimization, to optimize the execution of matrix expressions has also been outlined.
- It has further been shown that the FEAMOS approach provides high processing efficiency for application-critical operations, due to the availability of well-fitted and

tailored main-memory data representations and operations within AMOS. Processing efficiency is, however, further dependent on such issues as query optimization, data indexing and filtering, and an appropriate selection of data representation and processing location.

- A number of additional database technologies have been identified as being important for supporting efficient domain modelling and are acknowledged as potential topics for future research as described in Section 7.2.

The key database technologies that form the basis for this work are object-relational, extensible and main-memory database technologies. The extensibility should further cover the query language, the query processor, and the storage manager:

- The DBMS is object-relational due to the modelling and extensibility capabilities of the query language including overloaded and multi-directional functions.
- Extensibility of the query language provides domain-specific modelling of data and operations. The query processor also needs to be extensible to be able to cope with optimization and cost models for domain-specific foreign operations. Furthermore, the storage manager must be extensible to allow for incorporating tailored data representations, such as specialised matrix schemas.
- Main-memory is a prerequisite for being able to use the embedded DBMS approach and for supporting high processing efficiency.

Performance measures and comparisons between the original TRINITAS system and the integrated FEAMOS system show that the integrated system can provide competitive performance. The added DBMS functionality can be supplied without any major performance loss. In fact, under certain conditions the integrated system outperforms the original system and in general the DBMS provides better scaling performance. However, a full exploitation of the potential of the DBMS can only be accomplished by intelligent and careful design that takes advantage of DBMS facilities for filtering and accessing data in application operations.

It is argued here that the suggested architecture can form a promising alternative for the design and implementation of FEA applications and similar scientific and engineering applications. The present approach provides positive effects on both the external and the internal level. It facilitates integration of, or communication with, an engineering application with other parts of a global EIS system. By embedding a lightweight and extensible MM DBMS in an application you get a standardized query language for representing, managing, and exchanging domain data as well as access to generic software facilities for implementation of engineering applications without sacrificing execution efficiency. It will then be possible to achieve a global improvement in the efficiency of FEA software from the point of view of the developer, the maintainer, and the user. It might further be expected that this will result in the increased life-time of data and software and that the analysis quality can be enhanced.

Generally, database technology can play a similar and important role in the implementation of scientific and engineering applications of tomorrow, as it is currently doing in administrative applications. Specifically, we believe that database technology adaptable to the requirements of engineering applications such as OR DBMSs will play an important role in this area.

7.2 FUTURE WORK

A broad spectrum of potential research topics arises for future research within this field. A number of specific topics have been pointed out throughout this thesis.

From the FEA perspective, several of the ideas in this work can be further developed. The geometry representation could be extended with more advanced geometry representations, for example the ability to handle solids in the query language. The inclusion of more complex operations in the existing geometry model is also of interest, especially in combination with spatial indexing techniques.

As shown in Section 5.3.2, a query language can be highly convenient for expressing and extracting logically related data which exist in an FEA mesh and its relations to geometry, boundary conditions, and equation system. A more general applicability of these ideas is expected to provide improved results for the modelling of FEA data. To support the efficient treatment of various relationships, it would be of interest to study function materialisation techniques.

The work on the data source for numerical matrix algebra has revealed several future research topics, including optimization of matrix algebraic expressions, a combination of main-memory and secondary storage matrix representations, and the ability to express more complex algorithms in the query-language-based matrix algebra. This also includes the ability to express additional parts of the analysis process, such as calculating element stiffnesses and load components, and assembling the global stiffness matrix.

In the evaluation of analysis results, the ad hoc query capability can provide a powerful and flexible mechanism that can add functionality to conventional and “hard-coded” postprocessors. An issue for further research would be to investigate how the query language could be extended with graphical presentation primitives, as suggested in the thesis, to support the evaluation of large and distributed (spatially or temporally) data sets. The field of geographical information systems can probably provide valuable experience in this matter. The application of various data indexing techniques to provide processing efficiency is also a potential research area in this context.

Furthermore, future research in database technology for FEA applications should also consider the possibility of making comparisons with commercial FEA software, especially in the context of processing efficiency.

In general, it is of interest to study how more complex operations can be described in declarative query languages, such as AMOSQL. It has here been suggested that the extension of AMOSQL to preserve the uniqueness of sets and the order of sequences in query expressions should be investigated. For instance, it is expected that this can facilitate the formulation of geometry-related operations and queries.

From the DBMS perspective, optimization techniques and cost models for domain-specific operations should be further investigated. This includes the optimization of query expressions involving matrix operations and also more problem-related knowledge that takes advantage of certain problem characteristics to guide the execution.

Again, the applicability of general indexing techniques, such as spatial indexes, in FEA and provided by the DBMS need further investigation. Another research area of interest is the potential capability to use function materialisation techniques that could include cost models for storage space and processing and for handling the computation of function values. For instance, if the geometry discretisation operation could be supported by a mechanism of this kind, the FEA mesh could automatically be recomputed if the geometry has been changed since the last time mesh data were accessed.

The application of rules in FEA for controlling the analysis process or for automatic result evaluation have also been discussed in this work, but needs further study. However, earlier work by the author [152] has shown how rules in knowledge-based systems can be applied for similar purposes.

Using conventional database transactions and long-running transactions, such as sagas [155], to control the FEA process and database consistency has not been treated in this work but needs to be addressed if DBMS facilities such as concurrency and recovery are to be supported.

In another area, product data management, some activities have already been performed, Orsborn [120] [121]. An important ingredient within this area is the STEP standard [77] and we are currently engaged in providing AMOS with an EXPRESS [79] interface. This will, for example, provide a base for exchanging FEA data with other STEP-based applications.

Finally, other areas of interest include applying temporal query capabilities to express dynamic FEA problem classes, distributed database technology for storage and processing of FEA data, and query language-supported parallelisation of FEA processing.

8 REFERENCES

1. French, J. C., Jones, A. K., and Pfaltz, J. L., “Summary of the Final Report of the NSF Workshop on Scientific Database Management”, SIGMOD Record, v. 19 n. 4, December 1990, p. 32-40.

2. IEEE Computer Society, “*The Bulletin of the Technical Committee on Data Engineering*” *(TCDE)*, Special Issue on Scientific Databases, v. 93 n. 2, 1993.

3. Maier, D. and Vance, B., “A Call to Order”, Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Washington, DC, May 1993, p. 1-16.

4. DBMS, “A New Direction in DBMS”, Interview with Michael R. Stonebraker, DBMS, v. 7 n. 2, February 1994, p. 50-60.

5. Frank, M., “Object-Relational Hybrids”, DBMS, v. 8 n. 8, July 1995, p. 46-56.

6. Stonebraker, M, and Moore, D., “*Object-Relational DBMSs: The Next Great Wave*”, Morgan Kaufmann Publishers, Inc., 1996.

7. Orsborn, K., “Applying Next Generation Object-Oriented DBMS for Finite Element Analysis”, Proceedings of the 1st International Conference on Applications of Databases (ADB94), Vadstena, June 20-22, 1994, p. 215-233.

8. Fahl, G., Risch, T., and Sköld, M., “AMOS - An Architecture for Active Mediators”, The International Workshop on Next Generation Information Technologies and Systems (NGITS’ 93), Haifa, Israel, June 28-30, 1993, p. 47-53.

9. Flodin, S., Karlsson, J., Orsborn, K., Risch, T., Sköld, M., and Werner, M., “AMOS Users’s Guide”, EDSLAB, Linköping University, Linköping, March 1994.

10. Torstenfelt, B., Allestam, H., and Klarbring, A., “Shape Optimization Implemented in an Object-Oriented Finite Element Program Environment”, 6th Nordic Seminar on Computational Mechanics, Linköping, 1993.

11. Torstenfelt, B., “An Integrated Graphical System for Finite Element Analysis”, User’s Manual Version 2.0, LiTH-IKP-R-737, Linköping University, Linköping, January 1993.

12. Wiederhold, G., Risch, T., Rathmann, P., DeMichiel, L., Chaudhuri, S., Lee, B. S., Law, K. H., Barsalou, T., and Quass, D., “A Mediator Architecture for Abstract Data Access”, Tech. report STAN-CS-90-1303, Stanford University, Stanford, February, 1990.

13. Wiederhold, G. “Mediators in the Architecture of Future Information Systems”, IEEE Computer, March 1992, p.38-49.

14. Risch, T. and Wiederhold, G. “Building Adaptive Applications Using Active Mediators”, Proceedings of Database and Expert Systems Applications (DEXA ‘91), 1991.

15. Lyngbaek, P., “OSQL: A Language for Object Databases”, HPL-DTD-91-4, Hewlett-Packard Company, January 1991.

16. Melton, J. (ed.), ANSI SQL3 Papers SC21 N9463 - SC21 N9467, ANSI SC21 Secretariat, New York, U.S.A., 1995.

17. Cattell, R. G. G. (ed.), “*The Object Database Standard: ODMG-93, Release 1.2*”, Morgan Kaufmann Publishers, Inc., 1994.

18. Orsborn, K. and Risch, T., “Next Generation of O-O Database Techniques in Finite Element Analysis”, Proceedings of the 3rd International Conference on Computational Structures Technology (CST96), Budapest, Hungary, August 21-23, 1996, p. 121-136.

19. Flodin, S., Orsborn, K., and Risch, T., “Using Multi-Method Queries in Finite Element Analysis”, submitted to the 13th International Conference on Data Engineering, Birmingham, U.K., April 7-11, 1997.

20. Ottosen, N. and Petersson, H. “*Introduction to the Finite Element Method*”, Prentice Hall International Ltd., 1992.

21. Becker, E. B., Carey, G. F., and Oden, J. T., “*Finite Elements: An Introduction*”, Prentice-Hall, Inc., v. 1, Texas Finite Element Series, 1981.

22. Reddy, J. N., “*An Introduction to the Finite Element Method*”, MacGraw-Hill, Inc., 1985.

23. Hughes, J. T. H. “*The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*”, Prentice Hall International Ltd., 1987.

24. Chalfan, K. M., “An Integration Tool for Life-Cycle Engineering”, Knowledge Representation, 1986, p. 592-595.

25. Alsina, J., Fielding, J., and Morris, A., “ADROIT - An Expert System for Aircraft Design”, Aerogram, v. 4 n. 5, May 1987, p. 2-5.

26. Mitchell, A. R., Bryan, S. S., and Hall, M. D., “Design Engineering Technologies for Aerospace Vehicles”, Tech. report American Institute of Aeronautics and Astronautics, Inc., 1987.

27. Abelson, H., Eisenberg, M., Halfant, M., Katzenelson, J., Sacks, E., Sussman, G. J., Wis-

dom, J., and Yip, K., “Intelligence in Scientific Computing”, Communications of the ACM, v. 32 n. 5, May 1989, p. 546-562.

28. Forde, B. W. R., Russell, A. D., and Stiemer, S. F., “Object-Oriented Knowledge Frameworks”, Engineering with Computers, v. 5, 1989, p. 79-89.

29. Ahmed, S., Wong, A., Sriram, D., and Logcher, R., “Object-Oriented Database Management Systems for Engineering: A Comparison”, Journal of Object-Oriented Programming, v. 5 n. 3, June 1992, p. 27-44

30. Eastman, C.M., “The Contribution of Data Modeling to the Future Development of CAD/CAM Databases”, Proceedings of the 1991 ASME International Computers in Engineering Conference and Exposition, Santa Clara, CA, USA, 1991 August 18-22. Published in Engineering Databases: An Enterprise Resource, ASME, New York, NY, USA, 1991, p. 49-54.

31. Beck, R., Cue, R., and Schricker, V., “Engineering Databases - Current Technology and Future Directions”, Proceedings of the 1992 Pressure Vessels and Piping Conference, New Orleans, LA, USA, 1992 June 21-25. Published in Computer Technology - 1992 - Advances and Applications, ASME, Pressure Vessels and Piping Division, PVP v. 234, New York, NY, USA, 1992, p. 73-82.

32. Samaras, G., Spooner, D., and Hardwick, M., “Query Classification in Object-Oriented Engineering Design Systems”, Computer-Aided Design, v. 26 n. 2, February 1994, p. 127-136.

33. Mackerle, J. and Orsborn, K., “Expert Systems for Finite Element Analysis and Design Optimization - A Review”, Engineering Computations, v. 5 n. 2, 1988, p. 90-102.

34. Forde, B. W. R. and Stiemer, S. F., “Knowledge-Based Control for Finite Element Analysis”, Engineering with Computers, v. 5 n. 3-4, 1989, p. 195-204.

35. Ramirez, M. R. and Belytschko, T., “An Expert System for Setting Time Steps in Dynamic Finite Element Programs”, Engineering with Computers, v. 5 n. 3-4, 1989, p. 205-219.

36. Shephard, M. S., Baehmann, P. L., Georges, M. K., and Korngold, E. V., “Framework for the Reliable Generation and Control of Analysis Idealizations”, Computer Methods in Applied Mechanics and Engineering, v. 82, 1990, p. 257-280.

37. Tworzydlo, W. W. and Oden, J. T., “Towards an Automated Environment in Computational Mechanics”, Computer Methods in Applied and Engineering, v. 104, 1993, p. 87-143.

38. Baugh, J. W., and Rehak, D. R., “Object-Oriented Design of Finite Element Programs”, Computer Utilization in Structural Engineering Proceedings of the Sessions at Structures Congress ‘89, San Francisco, CA, USA, May 1-5, 1989, p. 91-100.

39. Fenves, G. L., “Object-Oriented Programming for Engineering Software Development”, Engineering with Computers, v. 6, 1990, p. 1-15.

40. Forde, B. W. R., Foschi, R., and Stiemer, S. F., “Object-Oriented Finite Element Analysis”, Computers & Structures, v. 34 n. 3, 1990, p. 355-374.

41. Filho, J. S. R. A., and Devloo, P. R. B., “Object-Oriented Programming in Scientific Computations: the Beginning of a New Era”, Engineering Computations, v. 8, 1991, p. 81-87.

42. Dubois-Pelerin, Y., Zimmermann, T., and Bomme, P., “Object-Oriented Finite Element Programming: II. A Prototype Program in Smalltalk”, Computer Methods in Applied and Engineering, v. 98, 1992, p. 361-397.

43. Williams, J. R., Lim, D., and Gupta, A., “Software Design of Object Oriented Discrete Element Systems”, Proceedings of the Third International Conference on Computational Plasticity, Barcelona, Spain, April 6-10, 1992, p. 1937-1947.

44. Scholz, S. -P., “Elements of an Object-Oriented FEM++ Program in C++”, Computers & Structures, v. 43 n. 3, May 1992, p. 517-529.

45. Baugh, J. W. and Rehak, D. R., “Data Abstraction in Engineering Software Development”, Journal of Computing in Civil Engineering, v. 6 n. 3, July 1992, p. 282-301.

46. Mackie, R. I., “Object Oriented Programming of the Finite Element Method”, International Journal for Numerical Methods in Engineering, v. 35 n. 2, August 1992, p. 425-436.

47. Ross, T. J., Wagner, L. R., and Luger, G. F., “Object-Oriented Programming for Scientific Codes. II: Examples in C++”, Journal of Computing in Civil Engineering, v. 6 n. 4, October 1992, p. 497-514.

48. Raphael, B. and Krishnamoorthy, C. S., “Automating Finite Element Development Using Object Oriented Techniques”, Engineering Computations, v. 10 n. 3, June 1993, p. 267-278.

49. Yu, G. and Adeli, H., “Object-Oriented Finite Element Analysis Using EER Model”, Journal of Structural Engineering, v. 119 n. 9, September 1993, p. 2763-2781.

50. Hoffmeister, P., Zahlten, W., and Krätzig, W. B., “Object-Oriented Finite Element Modeling”, Proceedings of the 5th International Conference on Computing in Civil and Building Engineering (V-ICCCBE), Anaheim, CA, USA. Published by ASCE, New York, USA, 1993, p. 537-544.

51. Arruda, R. S., Landau, L., and Ebecken, N. F. F., “Object-Oriented Structural Analysis in a Graphical Environment”, In Artificial Intelligence and Object-Oriented Approaches for Structural Engineering, Topping, B. H. V. and Papdrakakis, M. (eds.), Civil-Comp Press, 1994, p. 129-138.

52. Devloo, P. R. B., “Efficiency Issues in an Object-Oriented Programming Environment”, In Artificial Intelligence and Object-Oriented Approaches for Structural Engineering, Topping, B. H. V. and Papdrakakis, M. (eds.), Civil-Comp Press, 1994, p. 147-151.

53. Eyheramendy, D. and Zimmermann, T., “Object-Oriented Finite Element Programming: Beyond Fast Prototyping”, In Artificial Intelligence and Object-Oriented Approaches for Structural Engineering, Topping, B. H. V. and Papdrakakis, M. (eds.), Civil-Comp Press, 1994, p. 121-127.

54. Gajewski, R. R., “An Object-Oriented Approach to Finite Element Programming”, In Artificial Intelligence and Object-Oriented Approaches for Structural Engineering, Topping, B. H. V. and Papdrakakis, M. (eds.), Civil-Comp Press, 1994, p. 107-113.

55. Ju, J. and Hosain, M. U., “Substructuring Using an Object-Oriented Approach”, In Artificial Intelligence and Object-Oriented Approaches for Structural Engineering, Topping, B. H. V. and Papdrakakis, M. (eds.), Civil-Comp Press, 1994, p. 115-120.

56. Shepherd, D. A., and Lefas, I. D., “The Use of an Object-Oriented Language in the Development of Structural Engineering Programs”, In Artificial Intelligence and Object-Oriented Approaches for Structural Engineering, Topping, B. H. V. and Papdrakakis, M. (eds.), Civil-Comp Press, 1994, p. 153-157.

57. Langtangen, H. P., “DIFFPACK: Software for Partial Differential Equations”, Tech. report

STF33 A94020, SINTEF, Oslo, Norway, March, 1994.

58. Cardona, A., Klapka, I., and Geradin, M., “Design of a New Finite Element Environment”, Engineering Computations, v. 11, 1994, p. 365-381.

59. Zeglinski, G. W., Han, R. P. S., and Aitchison, P., “Object Oriented Matrix Classes for Use in a Finite Element Code Using C++”, International Journal for Numerical Methods in Engineering, v. 37, 1994, p. 3921-3937.

60. Lu, J., White, D. W., Chen, W. -F., and Dunsmore, H E., “A Matrix Class Library in C++ for Structural Engineering Computing”, Computers & Structures v. 55 n. 1, 1995, p. 95-111.

61. Grant, P. W., Sharp, J. A., Webster, M. F., and Zhang, X., “Some Issues in a Functional Implementation of a Finite Element Algorithm”, 6th International Conference on Functional Programming Languages and Computer Architectures, Copenhagen, June 1993, p. 12-17.

62. Yeh, C. -P., Fulton, R. E., and Peak, R. S., “A Prototype Information Integration Framework for Electronic Packaging”, ASME Winter Annual Meeting, Atlanta, GA, USA, 1991 December 1-6, Published by ASME, New York, NY, USA, 91-WA-EEP-43, 1991, p. 1-8.

63. Felippa, C. A., “Implementation of Scientific Data Management in Computational Mechanics: Personal Experiences”, In “*State-of-the Art Surveys on Computational Mechanics*”, ISBN: 0-7918-0303-1, 1989, p. 469-491.

64. Dopker, B., Murray, P., and Choong, F. N., “Object Oriented Data Base and Application Management System for Integrated”, Interdisciplinary Mechanical System Simulation, Published in Mechanical systems analysis, design and simulation, ASME, Design engineering division, DE v. 3(3), New York, NY, USA, 1989, p. 81-87.

65. Santana, O., Chia, B. T., Coulomb, J. L., and Iafrate, J. P., “Data Bases for CAD Applications”, Third Biennial Conference on Electromagnetic Field Computation, Washington, DC, USA, 1988 December 12-14. Published in IEEE Transactions on Magnetics v. 25 n. 4, July 1989, p. 2956-2958.

66. Myers, K. W., “New Tools for FEA. Solving the Data Management Problem”, Finite Element Analysis, Computer Applications, and Data Management - Presented at the 1990 Pressure Vessels and Piping Conference, Nashville, TN, USA, 1990 June 17-21. Published in ASME, Pressure Vessels and Piping Division, PVP v. 185, New York, NY, USA, 1990, p. 37-41.

67. Xingjian, Y., “Database Design Technique for Finite Element Analysis”, Second World Congress on Computational Mechanics - WCCM II, Stuttgart, Germany, 1990 August 27-31. Published in Computer Methods in Applied Mechanics and Engineering v. 91 n. 1-3, October 1991, p. 1357-1364.

68. Spainhour, L. K., Patton, E. M., Burns, B. P., Rasdorf, W. J., and Collier, C. S., “Computer-Aided Analysis System with DBMS Support for Fiber-reinforced Thick Composite Materials”, Proceedings of the 1991 ASME International Computers in Engineering Conference and Exposition, Santa Clara, CA, USA, 1991 August 18-22. Published in Engineering Databases: An Enterprise Resource, ASME, New York, NY, USA, 1991, p. 37-48.

69. Krishnamoorthy, C. S. and Umesh, K. R., “Adaptive Mesh Refinement for Two-dimensional Finite Element Stress Analysis”, Indian Inst. of Technology, Madras, India. Published in Computers and Structures v. 48 n. 1, July 1993, p. 121-133.

70. Pepper, D. W. and Marino, J. A., “Object Oriented Relational Database for Assessing Radioactive Material Transport”, Proceedings of the 4th Annual International Conference on High Level Radioactive Waste Management, Las Vegas, NV, USA. Published by ASCE, New York, NY, USA, 1993, p. 1187-1193.

71. Magnin, H. and Coulomb, J. L., “Towards a Distributed Finite Element Package for Electromagnetic Field Computation”, 5th Biennial IEEE Conference on Electromagnetic Field Computation, Claremont, CA, USA. Published in IEEE Transactions on Magnetics v. 29 n. 2, March 1993, p. 1923-1926.

72. Yang, J. and Yang, N., “A Brief Review of FEM Software Technique”, Advances in Engineering Software v. 17 n. 3, 1993, p. 195-200.

73. Baker, P., “Integrated Approach to Finite Element Analysis of Advanced Composite Structures”, Computer-Aided Design, v. 21 n. 7, September 1989, p. 441-446.

74. Felippa, C. A., “Database Management in Scientific Computing - I. General Description”, Computers and Structures, v. 10 n. 1, 1979, p. 53-61.

75. Bergman, G., Oldenburg, M., and Jeppsson, P., “Integration of a Product Design System and Nonlinear Finite Element Codes via a Relational Database”, Engineering Computations, v. 12, 1995, p. 439-449.

76. Ketabchi, M. A., Mathur, S., Risch, T., and Chen, J., “Comparative Analysis of RDBMS and OODBMS: A Case Study”, IEEE Computer Soc. International Conference 35, San Francisco, 1990. Digest of papers/Compcon spring 90, February 26 - March 2, 1990, p. 528-537.

77. ISO 10303-1, “Product Data Representation and Exchange - Part 1: Overview and Fundamental Principles”, ISO 10303-1, International Organization for Standardization, 1992.

78. ISO 10303-104, “Part 104 - Integrated Application Resources: Finite Element Analysis”, ISO 10303-104, International Organization for Standardization, 1992.

79. ISO 10303-11, “Part 11 - Description Methods: The EXPRESS Language Reference Manual”, ISO 10303-11, International Organization for Standardization, 1992.

80. Elmasri, R. and Navathe, S. B., “*Fundamentals of Database Systems*”, 2nd ed., The Benjamin/Cummings Publishing Company, Inc., 1994.

81. Loomis, M. E. S., “*The Database Book*”, Macmillan Publishing Company, 1990.

82. Melton, J. and Simon, A. R., “*Understanding the New SQL: a Complete Guide*”, Morgan Kaufmann Publishers, Inc., 1993.

83. Date, C. J., “*An Introduction to Database Systems*”, v. 1, 5th ed., Addison-Wesley Publishing Company, Inc., 1990.

84. Tsichritzis, D. and Klug, A. (eds.), “*The ANSI/X3/SPARC DBMS Framework*”, AFIPS Press, 1978.

85. Cattell, R. G. G., “*Object Data Management: Object-Oriented and Extended Relational Database Systems*”, Addison-Wesley Publishing Company, Inc., 1991 (reprinted with corrections 1992).

86. Flodin, S. and Risch, T., “Processing Object-Oriented Queries with Invertible Late Bound Functions”, Proceedings of the 1995 Conference on Very Large Databases, September 1996, p. 335-344.

87. Flodin, S., “Efficient Management of Object-Oriented Queries with Invertible Late Bound Functions”, Licentiate Thesis LiU-Tek-Lic 1996:03, Linköping University, Linköping, February, 1996.

88. Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., and Zdonik, S., “The Object-Oriented Database System Manifesto”, In Kim, W., Nicolas, J-.M., and Nishio, S., (eds.), Proceedings of the First International Conference on Deductive and Object-Oriented Databases (DOOD), Elsevier Science Publishers, Amsterdam, 1989, p. 40-57.

89. Stonebraker, M, Rowe, L. A., Lindsay, B., Gray, J., Carey, M., Brodie, M., Bernstein, P., and Beech, D., “Third-Generation Database System Manifesto”, SIGMOD Record, v. 19 n. 3, September 1990, p. 31-44.

90. Kim, W. “*Modern Database Systems: The Object Model, Interoperability, and Beyond*”, Addison-Wesley Publishing Company, 1995.

91. Carey, M. (ed.), Special issue on extensible database systems, Database Engineering, v. 10 n. 2, June 1987.

92. Carey, M. and Haas, L., “Extensible Database Management Systems”, SIGMOD Record, v. 19 n. 4, December 1990, p. 54-60.

93. Werner, M., “Multidatabase Integration Using Polymorphic Queries and Views”, Licentiate Thesis LiU-Tek-Lic 1996:11, Linköping University, Linköping, March 1996.

94. McPherson, J. and Pirahesh, H., “An Overview of Extensibility in Starburst”, Database Engineering, v. 10 n. 2, June 1987, p. 92-99.

95. Stonebraker, M., Anton, J., and Hirohama, M., “Extendability in Postgres”, Database Engineering, v. 10 n. 2, June 1987, p. 76-83.

96. Goldhirsh, D. and Orenstein, J., “Extensibility in the PROBE Database System”, Database Engineering, v. 10 n. 2, June 1987, p. 84-91.

97. Woelk, D. and Kim, W., “An Extensible Framework for Multimedia Information Management”, Database Engineering, v. 10 n. 2, June 1987, p. 115-121.

98. Batory, D. S., “Principles of Database Management System Extensibility”, Database Engineering, v. 10 n. 2, June 1987, p. 100-106.

99. Carey, M. and DeWitt, D., “An Overview of the EXODUS Project”, Database Engineering, v. 10 n. 2, June 1987, 107-114.

100. Dewitt, D. J., Katz, R. H., Olken, F., Shapiro, L. D., Stonebraker, M. R., and Wood, D., “Implementation Techniques for Main Memory Database Systems”, SIGMOD Record, v. 14 n. 2, 1984, p. 1-8.

101. Eich, M. H. (ed.), “Main-Memory Databases: Current and Future Research Issues (foreword)”, Special section on main-memory databases, IEEE Transactions on Knowledge and Data Engineering, v. 4 n. 6, December 1992.

102. Garcia-Molina, H.and Salem, K. “Main-Memory Database Systems: an Overview”, IEEE Transactions on Knowledge and Data Engineering, v. 4 n. 6, December 1992, p. 509-516.

103. Litwin, W., and Risch, T., “Main Memory Oriented Optimization of OO Queries Using Typed Datalog with Foreign Predicates”, IEEE Transactions on Knowledge and Data Engineering, v. 4 n. 6, December 1992, p. 517-528.

104. Listgarten, S. and Neimat, M-.A., “Modeling Costs for a MM-DBMS”, Proceedings of the

1st International Workshop on Real-Time Databases: Issues and Applications, Newport Beach, CA, USA, March 7-8, 1996, p. 77-83.

105. Heytens, M., Listgarten, S., Neimat, M.-A., and Wilkinson, K., "Smallbase: A Main-Memory DBMS for High-Performance Applications", Research report, Database Technology Department, Hewlett-Packard Laboratories, September 1995.

106. Özsu, M. T. and Valduriez, P., "*Principles of Distributed Database Systems*", Prentice-Hall, Inc., 1991.

107. Widom, J. and Ceri, S., "*Active Database Systems: Triggers and Rules for Advanced Database Processing*", Morgan Kaufmann Publishers, Inc., 1996.

108. Risch, T., and Sköld, M., "Active Rules Based on Object-Oriented Queries", IEEE Data Engineering (special issue on active databases), v. 15 n. 1-4, December, 1992, p. 27-30.

109. Sköld, M., "Active Rules Based on Object Relational Queries - Efficient Change Monitoring Techniques", Licentiate Thesis LiU-Tek-Lic 1994:38, Linköping University, Linköping, September 1994.

110. Sköld, M. and Risch, T., "Using Partial Differencing for Efficient Monitoring of Deferred Complex Rule Conditions", Proceedings of the 12th International Conference on Data Engineering (ICDE'96), New Orleans, Louisiana, February 26 - March 1, 1996, 392-401.

111. Korth, H. F. and Silberschatz, A. "*Database System Concepts*", 2nd ed., McGraw-Hill, Inc., 1991.

112. Ullman, J. D., "*Principles of Database and Knowledge-Base Systems*", v. 1, Computer Science Press, Inc., 1988.

113. Arikawa, M., Kawakita, H., and Kambayashi, Y., "Dynamic Maps as Composite Views of Varied Geographic Database Servers", Proceedings of the 1st International Conference on Applications of Databases (ADB94), Vadstena, Sweden, June 20-22, 1994, p. 142-157.

114. Kemp, G. J. L., Jiao, Z., Gray, P. M. D., and Fothergill, J. E., "Combining Computation with Database Access in Biomolecular Computing", Proceedings of the 1st International Conference on Applications of Databases (ADB94), Vadstena, Sweden, June 20-22, 1994, p. 317-335.

115. Chandra, R. and Segev, A., "Using Next Generation Databases to Develop Financial Applications", Proceedings of the 1st International Conference on Applications of Databases (ADB94), Vadstena, Sweden, June 20-22, 1994, p. 190-203.

116. Moss, E. (ed.), Special issue on emerging object query standards, IEEE Data Engineering, v. 17 n 4., December, 1994.

117. Bancilhon, F., Delobel, C., and Kanellakis, P. (eds.), "*Building an Object-Oriented Database System: The Story of O_2*", Morgan Kaufmann Publishers, Inc., 1992.

118. Fishman, D. H., Annevelink, J., Chow, E. Connors, T., Davis, J. W., Hasan, W. Hoch, C. G., Kent, W., Leichner, S., Lyngbaek, P., Mahbod, B., Neimat, M. A., Risch, T., Shan, M. C., and Wilkinson, W. K., "Overview of the Iris DBMS", in Kim, W., Lochovsky, F. H. (eds.): *Object-Oriented Concepts, Databases, and Applications*, ACM Press, Addison-Wesley, 1989, p. 219-250.

119. Shipman, D. W., "The Functional Data Model and the Data Language DAPLEX", ACM TODS, v. 6, n. 1, March 1981, p. 140-173.

120. Orsborn, K., “Modeling of Product Data Using an Extensible O-O Query Language”, LiTH-IDA-R-93-15, Linköping University, Linköping, May 1993.

121. Orsborn, K., “Management of Product Data Using an Extensible Object-Oriented Query Language”, accepted at the Sixth International Conference on Data and Knowledge Systems for Manufacturing and Engineering (DKSME ‘96), Tempe, Arizona, October 24-25, 1996.

122. Takizawa, M., “Distributed Database System JDDBS”, JARECT Computer Science & Technologies 7, OHMSHA & North Holland, 1983, p. 262-283.

123. Fahl, G., “Object Views of Relational Data in Multidatabase Systems”, Licentiate Thesis LiU-Tek-Lic 1994:32, Linköping University, Linköping, June 1994.

124. Fahl, G. and Risch, T., “Query Processing Over Object Views of Relational Data“, (to be published in VLDB Journal).

125. Karlsson, J. S., An Implementation of Transaction Logging and Recovery in a Main Memory Resident Database System, Masters Thesis LiTH-IDA-Ex-94-04, Department of Computer and Information Science, Linköping University, Linköping, Sweden, 1995.

126. Näs, J., “Randomized optimization of object-oriented queries in a main-memory database management system”, Masters Thesis LiTH-IDA-Ex-93-25, Department of Computer and Information Science, Linköping University, Linköping, Sweden, 1993.

127. Abiteboul, S., and Bonner, A., “Objects and Views”, Proceedings of the ACM SIGMOD Conference, 1991, p. 238-247.

128. Flodin, S., Karlsson, J., Risch, T., Sköld, M., and Werner, M., “AMOS System Manual”, EDSLAB, Linköping University, Linköping, June 1996.

129. Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenny, A., Ostrouchov, S., and Sorensen, D., “LAPACK Users’ Guide”, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1995.

130. Dongarra, J. J., Pozo, R., and Walker, D. W., “An Object-Oriented Design for High Performance Linear Algebra on Distributed Memory Architectures”, Proceedings of the Object-Oriented Numerics Conference (OONSKI), Sunriver, Oregon, May 26-27, 1993.

131. Dongarra, J. J., Pozo, R., and Walker, D. W., “LAPACK++: A Design Overview of Object-Oriented Extensions for High Performance Linear Algebra”, Proceedings of Supercomputing ‘93, IEEE Press, 1993, p. 162-171.

132. Barton, J. J. and Nackman, L. R., “Wrapping LAPACK in Objects”, C++ Report, v. 7 n. 5, June 1995, p. 50-53.

133. Sarawagi, S. and Stonebraker, M., “Efficient Organization of Large Multidimensional Arrays”, Proceedings of 1994 IEEE 10th International Conference on Data Engineering, Houston, TX, USA, 14-18 February, 1994. p. 328-36.

134. Maier, D. and Hanson, D. M., “Bambi Meets Godzilla: Object Databases for Scientific Computing”, Proceedings of the Seventh International Working Conference on Scientific and Statistical Database Management, Charlottesville, Virginia, USA, September 28-30, 1994, p. 176-184.

135. Vandenberg, S. L. and DeWitt, D. J., “Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance”, Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data, Denver, Colorado, June 1991, p. 158-167.

136. Libkin, L., Machlin, R., and Wong, L., “A Query Language for Multidimensional Arrays: Design, Implementation, and Optimisation Techniques”, Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996. p. 228-239

137. Rotem, D. and Zhao, J. L., “Extendible Arrays for Statistical Databases and OLAP Applications”, Proceedings of the Eight International Conference on Scientific and Statistical Database Management, Stockholm, Sweden, June 18-20, 1996, p. 108-117.

138. Seamons, K. E., Chen, Y., Winslett, M., and Cho, Y., “Persistent Array Access Using Server-Directed I/O”, Proceedings of the Eight International Conference on Scientific and Statistical Database Management, Stockholm, Sweden, June 18-20, 1996, p. 98-107.

139. Golub, G. H. and van Loan, C. F., “*Matrix Computations*”, 2nd ed., The John Hopkins University Press, 1989.

140. Carey, G. F. and Oden, J. T., “*Finite Elements: Computational Aspects*”, Prentice-Hall, Inc., v. 3, Texas Finite Element Series, 1984.

141. Cook, R. D., “*Concepts and Applications of Finite Element Analysis*”, 3rd ed., John Wiley & Sons, Inc., 1989.

142. Wolniewicz, R., and Graefe, G., “Algebraic Optimization of Computations over Scientific Databases”, Proceedings of the 19th VLDB Conference, Dublin, Ireland, August 24-27, 1993, p. 13-24.

143. Wyle, C. R. and Barret, L. C., “*Advanced Engineering Mathematics*”, 5th ed., MacGraw-Hill, 1985.

144. Råde, L. and Westergren, B., “*Beta: Mathematics Handbook*”, 2nd ed., Studentlitteratur and Chartwell-Bratt, 1990.

145. Cole, R. L., and Graefe, G., “Optimization of Dynamic Query Evaluation Plans”, SIGMOD, 1994, p. 150-160.

146. Stroustrup, B., “*The C++ Programming Language*”, 2nd ed., Addison-Wesley Publ. Comp., 1993.

147. Chambers, C. and Leavens, G. T., “Typechecking and Modules for Multimethods”, ACM Transactions on Programming Languages and Systems, v. 17 n. 6, November 1995, p. 805-843.

148. Finnigan, P. M., Kela, A., and Davis, J. E., “Geometry as a Basis for Finite Element Automation”, Engineering with Computers v. 5, 1989, p. 147-160.

149. Kemper, A., Kilger, C., and Moerkotte, G., “Function Materilization in Object Bases: Design, Realization, and Evaluation”, IEEE Transactions on Knowledge and Data Engineering, v. 6 n. 4, August 1994, p. 587-608.

150. Orsborn, K., “Integration of Deeper Models for Fracture Analysis into Xfrac”, LiTH-IKP-R-542, Linköping University, Sweden, 1988.

151. Orsborn, K., “Using Knowledge-Based Techniques in Systems for Structural Design. Computers & Structures v. 40, n. 5, 1991, p. 1203-1211.

152. Orsborn, K. “Structural Design Systems Using Knowledge-Based Techniques: Applications to Damage Tolerance Design of Aircraft Structures” Licentiate Thesis n. 400, Linköping University, Sweden, October, 1993.

153. Brodie, M. L. and Mylopoulos, J. (eds.), “*On Knowledge Base Management Systems: Integrating Artificial Intelligence and Database Technologies*”, Springer-Verlag, 1986.

154. Schenk, D. A. and Wilson, P. R., *“Information Modeling: The EXPRESS Way”*, Oxford University Press, Inc., 1994.

155. Garcia-Molina, H. and Salem, K., “Sagas”, Proceedings of ACM SIGMOD 1987 Annual Conference, San Francisco, May 27-29, 1987, p. 249-259.

APPENDIX A: TRINITAS CONCEPTS

The subsequent list cover the main FEA-related concepts currently represented in TRINITAS. It is not claimed that this list is complete since TRINITAS is continuously developed and additional functionality is added.

GEOMETRY CONCEPTS

point

curve

- straight line
- circular segment
- parabola
- bezier cubic segment

surface

- triangular
- quadrangular
- polygon

volume

- 1D cylinder
- 2D constant thickness
  - triangular
  - quadrangular
  - plane polygon

- axi triangular torus
- axi quadrangular torus
- axi polygonic torus

- 3D tetrahedron
- 3D pentahedron
- 3D hexahedron
- 3D extruded polyhedron

DOMAIN PROPERTY CONCEPTS

material
- linear elastic isotropic

BOUNDARY PROPERTY CONCEPTS

force
- point load
- line load
- surface load
- volume load
- nodal load

fixed displacements
- fixed point
- fixed line
- fixed surface
- fixed node

prescribed displacements
- prescribed point
- prescribed line
- prescribed surface
- prescribed node

fixed temperature
- temperature on lines

convective heat transfer on lines

fixed flux on lines

MESH CONCEPTS

> 2D linear bar
> 2D nonlinear bar
> 2D constant strain triangle
> 2D linear strain triangle
> 2D bilinear quadrangle
> 2D quadratic lagrange
> 2D quadratic serendipity
> 3D trilinear hexahedral

TIME CONCEPTS

time domain
time intensity function

ANALYSIS CLASS CONCEPTS

linear static stress analysis

> optimization
>
> minimum weight
> maximum stiffness
> min(max) stress

linear buckling analysis

linear dynamic stress analysis

> eigenvalue
> transient analysis

steady state thermal analysis

transient thermal analysis

nonlinear static stress analysis

phase transformation analysis

quench simulation

EVALUATION CONCEPTS

node values
- reaction force
- displacement
- applied load
- sigma xx
- sigma yy
- sigma xy
- von mises

graph

- entity along line
- sequence of nodes
- function of time
- versus another entity
- external unit

APPENDIX B: FEAMOS DOMAIN MODEL

This list provides an overview to the important parts of the current FEA domain model in FEAMOS.

GENERAL TYPES

```
named_object;
```

GENERAL FUNCTIONS

```
name
construct
initiate
destruct
exist_object
get_object_type
object_named
remove_object
rename_object
coerce
objects_to_names
names_to_objects
```

GEOMETRIC TYPES

```
geometric_object
   volume
      D_1_volume
         constant_cross_section_volume
      D_2_volume
         triangular_section_volume
         quadrangular_section_volume
         polygon_section_volume
         triangular_torus_volume
         quadrangular_torus_volume
         polygon_polygon_volume
      D_3_volume
         tetrahedron_volume
         pentahedron_volume
         hexahedron_volume
         extruded_polyhedron_volume
   surface
      triangular_surface
      quadrangular_surface
      cylindrical_surface
      polygon_surface
   curve
      straight_line
      parabola_cubic_section
      bezier_cubic_segment
      arc
   point
```

GEOMETRY FUNCTIONS

Volume functions

```
exists_volume
volume_named
face_vector
faces
get_face_vector
set_face_vector
no_of_elements
no_of_nodes
element_set_name
node_set_name
nodes_per_element
material_name
thickness
cross_section
node_derivatives
```

```
element_type
get_volume_attributes
set_volume_attributes
initiate
create_triangular_volume
create_quadrangular_volume
create_polygon_volume
remove_volume
```

Surface functions

```
exists_surface
surface_named
edge_vector
edges
get_edge_vector
set_edge_vector
initiate
create_triangular_surface
create_quadrangular_surface
create_cylindrical_surface
create_polygon_surface
remove_surface(charstring ch) -> boolean
```

Curve functions

```
exists_curve
curve_named
get_curve_type
vertex_vector
vertices
get_vertex_vector
set_vertex_vector
division
get_division
set_division
density
get_density
set_density
create_straight_line
create_parabola_cubic_section
create_bezier_cubic_segment
create_arc
remove_curve
```

Point functions

```
exists_point
point_named
```

```
position
get_position
set_position
x_coordinate
y_coordinate
z_coordinate
create_point
remove_point
```

FINITE ELEMENT TYPES

```
fea_object
   element
      volume_element
      surface_element
      curve_element
   node
   load
      line_load
      point_load
   displacements
      fixed_displacements
```

FINITE ELEMENT FUNCTIONS

Mesh functions

```
face_vector
faces
edge_vector
edges
edges
node_vector
nodes
volume
surface
curve
point
```

Load functions

```
curve
intensity
```

Displacement functions

```
curve
```

ARRAY TYPES

```
array
   iarray
   farray
   darray
```

ARRAY FUNCTIONS

```
construct
initialise
destruct
name
array_named
size
ref
set
```

MATRIX TYPES

This overview of matrix types are slightly compacted is a view that shows the number of types for each basic matrix type, representation scheme, and matrix data type. Matrix names have been shortened to make the overview simpler.

```
rectangular matrix
   regular
      imatrix
         ...
      fmatrix
         row
         column
         square
            symmetric
               diagonal
            triangular
               upper triangular
                  upper unit triangular
               lower triangular
                  lower unit triangular
      dmatrix
         ...

   skyline
      imatrix
         ...
      fmatrix
         quadratic
            symmetric
            triangular
```

```
                 upper triangular
                    upper unit triangular
                 lower triangular
                    lower unit triangular
      dmatrix
         ...

   sparse
      imatrix
         ...
      fmatrix
         ...
      dmatrix
         ...
```

MATRIX FUNCTIONS

```
construct
initialise
destruct
name
size
ref
set
rows
columns
plus
minus
times
quotient
transpose
factorise

dindex (for skyline matrices)
```

TRINITAS MATRIX TYPE

```
trinitas_matrix
```

TRINITAS MATRIX FUNCTIONS

```
construct
initiate
rows
columns
format
precision
get_trinitas_matrix_attributes
```

APPENDIX C: FEAMOS FOREIGN FUNCTIONS

This appendix presents an overview of the existing implementations of foreign functions for different binding patterns for the matrix data source. Where overloaded function implementations exist for different matrix schemes it is indicated. In addition, functions are overloaded, when relevant, for different matrix data types. These variants are not included in this list.

PLUS

$$\mathbf{A}^{b}_{rect} + \mathbf{B}^{b}_{rect} = \mathbf{C}^{f}_{rect}$$

$$\mathbf{A}^{b}_{rect} + \mathbf{B}^{f}_{rect} = \mathbf{C}^{b}_{rect}$$

$$\mathbf{A}^{f}_{rect} + \mathbf{B}^{b}_{rect} = \mathbf{C}^{b}_{rect}$$

MINUS

$\mathbf{A}^{b}_{rect} - \mathbf{B}^{b}_{rect} = \mathbf{C}^{f}_{rect}$ derived AMOSQL function

$\mathbf{A}^b_{rect} - \mathbf{B}^f_{rect} = \mathbf{C}^b_{rect}$ derived AMOSQL function

$\mathbf{A}^f_{rect} - \mathbf{B}^b_{rect} = \mathbf{C}^b_{rect}$ derived AMOSQL function

TIMES

Regular representation

$a^b \cdot \mathbf{B}^b_{rect} = \mathbf{C}^f_{rect}$

$a^b \cdot \mathbf{B}^f_{rect} = \mathbf{C}^b_{rect}$

$\mathbf{A}^b_{rect} \cdot \mathbf{B}^b_{rect} = \mathbf{C}^f_{rect}$

$\mathbf{A}^b_{diag} \cdot \mathbf{B}^b_{col} = \mathbf{C}^f_{col}$

$\mathbf{A}^b_{diag} \cdot \mathbf{B}^f_{col} = \mathbf{C}^b_{col}$

$\mathbf{A}^b_{uptri} \cdot \mathbf{B}^b_{col} = \mathbf{C}^f_{col}$

$\mathbf{A}^b_{uptri} \cdot \mathbf{B}^f_{col} = \mathbf{C}^b_{col}$

$\mathbf{A}^b_{uputri} \cdot \mathbf{B}^b_{col} = \mathbf{C}^f_{col}$

$\mathbf{A}^b_{uputri} \cdot \mathbf{B}^f_{col} = \mathbf{C}^b_{col}$

$\mathbf{A}^b_{diag} \cdot \mathbf{B}^b_{uptri} = \mathbf{C}^f_{uptri}$

$\mathbf{A}^b_{diag} \cdot \mathbf{B}^b_{uputri} = \mathbf{C}^f_{uputri}$

$$\mathbf{A}^b_{sym} \cdot \mathbf{B}^b_{col} = \mathbf{C}^f_{col}$$

$$\mathbf{A}^b_{sym} \cdot \mathbf{B}^f_{col} = \mathbf{C}^b_{col}$$

$$\mathbf{A}^b_{lowtri} \cdot \mathbf{B}^b_{col} = \mathbf{C}^f_{col}$$

$$\mathbf{A}^b_{lowtri} \cdot \mathbf{B}^f_{col} = \mathbf{C}^b_{col}$$

$$\mathbf{A}^b_{lowutri} \cdot \mathbf{B}^b_{col} = \mathbf{C}^f_{col}$$

$$\mathbf{A}^b_{lowutri} \cdot \mathbf{B}^f_{col} = \mathbf{C}^b_{col}$$

$$\mathbf{A}^b_{lowtri} \cdot \mathbf{B}^b_{uptri} = \mathbf{C}^f_{sym}$$

$$\mathbf{A}^b_{lowutri} \cdot \mathbf{B}^f_{uputri} = \mathbf{C}^b_{sym}$$

Skyline representation

$$\mathbf{A}^b_{uptri} \cdot \mathbf{B}^b_{col} = \mathbf{C}^f_{col}$$

$$\mathbf{A}^b_{uptri} \cdot \mathbf{B}^f_{col} = \mathbf{C}^b_{col}$$

$$\mathbf{A}^b_{uputri} \cdot \mathbf{B}^b_{col} = \mathbf{C}^f_{col}$$

$$\mathbf{A}^b_{uputri} \cdot \mathbf{B}^f_{col} = \mathbf{C}^b_{col}$$

$$\mathbf{A}^b_{diag} \cdot \mathbf{B}^b_{uptri} = \mathbf{C}^f_{uptri}$$

$$\mathbf{A}^b_{diag} \cdot \mathbf{B}^b_{uputri} = \mathbf{C}^f_{uputri}$$

$$\mathbf{A}^{b}_{sym} \cdot \mathbf{B}^{b}_{col} = \mathbf{C}^{f}_{col}$$

$$\mathbf{A}^{b}_{sym} \cdot \mathbf{B}^{f}_{col} = \mathbf{C}^{b}_{col}$$

$$\mathbf{A}^{b}_{lowtri} \cdot \mathbf{B}^{b}_{col} = \mathbf{C}^{f}_{col}$$

$$\mathbf{A}^{b}_{lowtri} \cdot \mathbf{B}^{f}_{col} = \mathbf{C}^{b}_{col}$$

$$\mathbf{A}^{b}_{lowutri} \cdot \mathbf{B}^{b}_{col} = \mathbf{C}^{f}_{col}$$

$$\mathbf{A}^{b}_{lowutri} \cdot \mathbf{B}^{f}_{col} = \mathbf{C}^{b}_{col}$$

$$\mathbf{A}^{b}_{lowtri} \cdot \mathbf{B}^{b}_{uptri} = \mathbf{C}^{f}_{sym}$$

$$\mathbf{A}^{b}_{lowutri} \cdot \mathbf{B}^{f}_{uputri} = \mathbf{C}^{b}_{sym}$$

QUOTIENT

$$\mathbf{A}^{b}_{rect} / b^{b} = \mathbf{C}^{f}_{rect}$$

TRANSPOSE

Regular representation

$$\mathbf{A}^{b}_{uptri} \leftrightarrow \mathbf{B}^{f}_{lowtri}$$

$$\mathbf{A}^{f}_{uptri} \leftrightarrow \mathbf{B}^{b}_{lowtri}$$

$$\mathbf{A}^{b}_{uputri} \leftrightarrow \mathbf{B}^{f}_{lowutri}$$

$$\mathbf{A}^{f}_{uputri} \leftrightarrow \mathbf{B}^{b}_{lowutri}$$

$\mathbf{A}^{b}_{lowtri} \leftrightarrow \mathbf{B}^{f}_{uptri}$	derived AMOSQL function
$\mathbf{A}^{f}_{lowtri} \leftrightarrow \mathbf{B}^{b}_{uptri}$	derived AMOSQL function
$\mathbf{A}^{b}_{lowutri} \leftrightarrow \mathbf{B}^{f}_{uputri}$	derived AMOSQL function
$\mathbf{A}^{f}_{lowutri} \leftrightarrow \mathbf{B}^{b}_{uputri}$	derived AMOSQL function

Skyline representation

The same variants as for the regular representation

FACTORISE

Regular representation

$$\mathbf{A}^{b}_{sym} \rightarrow \mathbf{B}^{f}_{uputri}\mathbf{C}^{f}_{diag}$$

Skyline representation

The same variant as for the regular representation

INDEX