Efficient Database Implementation of EXPRESS Information Models

Efficient Database Implementation of EXPRESS Information Models

by

David Thomas Loffredo

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: Computer Science

Approved by the
Examining Committee:

Martin Hardwick, Thesis Advisor

Cheng Hsu, Member

Mukkai S. Krishnamoorthy, Member

David L. Spooner, Member

Rensselaer Polytechnic Institute
Troy, New York

May 1998

Contents

List of Tables

List of Figures

Acknowledgments

After ten long years, it seems that I have built up quite a debt of gratitude.

First, on a professional level, I thank Alok Mehta for his work on the ObjectStore and Versant projects; Rick Kramer and Vijay Raghavan for their work on the ORACLE project; and Charles Gilman for exercising the SDAI binding used in the Versant project. Thanks also to Sun Jianhong and Jeff Young for their thorough review of this document.

Next on a personal level, I thank Dr. Martin Hardwick and Dr. Dave Spooner for their support and guidance over the last ten years.

So many people have provided continuous encouragement and leadership by example: my entire family; everyone at STEP Tools; my rose-buds: Dr. Alyce ("Jasmine") Faulstich-Brady and Dr. Don ("Begonia") Sanderson; the Party Consortium: Dr. Tom Bilodeau, Dr. Kurt Dittenberger, Dr. Fabio Guerinoni, Jay Hersh, and soon-to-be Dr. Dave Tonnesen; Drs. Dave McIntyre and Dave Berque; Dr. Ron Kelley, John Klym, and Danny Murray.

Paula Popson was literally the first person I met at RPI, and throughout the procession of centers — CICG, RDRC, DMI, and finally CAT/LIII — she has always watched out for me. Pam Paslow and Mary Johnson have also been there for me over the years.

To Dr. Mike DeMarco, who convinced me to go to graduate school and warned me about the associated “chased by monster” dreams.

To Marty and Dave, who have been there from kindergarden to the not-so-bitter end.

And finally to The Delightful Miss Krista for providing the motivation and encouragement to finish this damn thing.

Abstract

The research presented in this thesis describes a framework for database implementation of EXPRESS information models. EXPRESS models describe complex structures and correctness conditions for engineering activities, and are defined by the ISO-10303 Standard for Product Data Exchange (STEP). These models are a substantial engineering resource, and industry desires to use them to integrate design and manufacturing processes. Databases built around STEP models are essential because they provide content that integrated engineering processes understand.

The Standard Data Access Interface (SDAI) is a STEP API for EXPRESS-defined data. Prototypes have attempted to provide SDAI access by implementing each SDAI operation as one or more native operations directly upon the database. A direct binding can be costly, as it requires completely new software for each database. This work proposes several SDAI implementation architectures that offer alternatives to a direct binding.

To evaluate the real-world performance of implementations, this work defines a set of representative benchmarks on the STEP AP-203 information model. AP-203 contains information such as CAD geometry and product configuration that is common to all of the STEP models. The STEPStone benchmarks cover information that is modeled in an existence dependent style (PartStone, part versions), a navigational style (NURBStone, geometry), and a mix of the two (BOMStone, bill of material).

The results of timing experiments using these benchmarks are presented. The experiments evaluate the performance of direct-binding SDAI implementations built on relational and object-oriented databases, and examine the effect of various optimizations on binding performance. Analysis of the timing results provide the relative cost of access for each system, and allow us to determine when each implementation style will be most advantageous. In addition, these experiments provide insight about the use of SDAI access versus traditional access strategies.

1 Introduction

1.1 Motivation

Design and manufacturing companies want to integrate their engineering processes around product databases, but engineering databases are expensive and difficult to create. Integration around product databases can enable concurrent engineering — a process where multiple engineers work on different facets of a product concurrently [Winn88]. However, integrated product databases are not yet common in industry.

One reason for this is because engineering applications have unusually complex information models. These information models are complex because engineering applications manipulate simulations of the real world. Models for areas such as CAD geometry, tolerances, materials, and manufacturing plans are structurally and semantically rich. Applications are similarly complex, and are tightly bound to the models. Consequently, developers can only afford to build tools around successful models. To date, these have been the geometric models of popular CAD systems such as AutoCAD, CATIA, Pro/Engineer, and Unigraphics.

Often, the information models exist only as program language structures taken from a primary application, usually a CAD system. Without a well-defined model, subsequent applications must be modified whenever the primary application changes. In practice, only small, highly focused, applications are ever developed by anyone other than the primary application vendor.

The resulting situation is that only special-purpose databases, controlled by CAD vendors, are used to describe complex products. Manufacturers do not have any control over their product databases, which is clearly undesirable for strategic reasons. Also, applications that improve segments of a market cannot be applied to an industry that is locked into vertical applications. The dominance of special-purpose databases and vertically-integrated applications is a major reason why the general-purpose engineering database market remains small [Hard95b].

Industry has begun to address this problem by developing standard engineering information models. The ISO-10303 Standard for Product Data Exchange (STEP) contains formal descriptions of the information used by the engineering activities in a product lifecycle. These models are the result of significant investments of time and expertise, and represent the agreement of many interested parties on the scope, content, and correctness conditions of the information.

Documenting engineering information models requires a formalism that can handle complex structures and correctness conditions. The STEP models are written using the EXPRESS language. EXPRESS and other formalisms make it easier to describe an accurate information model, but do not dictate how the models should be implemented using various database technologies.

Because formal information models for engineering information are only now beginning to appear, the literature does not adequately address ways in which databases should provide structurally complex information to engineering applications. The research presented in this thesis addresses the issues surrounding this problem.

1.2 General Approach

The STEP information models have been used for file exchange between applications, but have not yet been widely used to define shared engineering databases. Exchange is done using a file format (Part 21) that can be quickly added to existing applications. Database implementations are expected to provide access to data using the Standard Data Access Interface (SDAI), a STEP API for EXPRESS-defined data.

The SDAI is a set of protocols, still under development, the goal of which is to reduce the cost of complex engineering applications by making them portable across different storage technologies. The SDAI protocols contain a description of the operations that must be provided (functional specification), and several bindings that describe how these operations are made available in different programming language environments.

Some prototype systems have attempted to provide SDAI access by implementing each SDAI operation as one or more native operations directly upon the database. This approach can be costly, as it requires completely new SDAI binding software for each database. Furthermore, the target database system may not handle the full range of the EXPRESS structures. Unsupported structures can often be simulated using other structures, but encoding and decoding data at run-time may reduce the performance of SDAI operations.

This work investigates the ways in which SDAI access to a database can be provided and proposes a framework that offers a range of implementation architectures. This framework introduces alternate architectures that permit more code reuse and offer different sets of capabilities.

We survey the implementation costs using systems built on a variety of databases. To gain insight into operational costs, we test a selection of real-world operations against systems based on Oracle and ObjectStore. Oracle is a relational system that holds data in normalized tables and relies on query-based access. ObjectStore is an object-oriented system that stores data in clusters and has an interface tuned for navigational access. Together, they represent both extremes of database systems used by engineering applications.

The operational tests are based on the STEP AP-203 information model. AP-203 contains information, such as CAD geometry and product configuration, that is common to all of the STEP models. A set of representative benchmarks on this model are presented. The STEPStone benchmarks cover information that is modeled in an existence-dependent style (PartStone, part versions), a navigational style (NURBStone, geometry), and a mix of the two (BOMStone, bill of material).

1.3 Results

The work presented in this thesis offers guidance to those who must implement engineering databases around complex models. In particular, this work presents a systematic survey of implementation architectures, a set of benchmarks that simulate how engineering applications access a database, and recommendations drawn from experiments with a number of database systems. These results use EXPRESS, the SDAI, and AP-203, but could be applied to other modeling formalisms or engineering areas.

In summary, this work makes the following contributions:

- Definition of a framework for database implementation of EXPRESS models. These implementation architectures can be grouped into upload/download, cached, and direct bindings. The implementation cost for the architectures are illustrated using systems built on a variety of databases.

- Definition of a representative set of benchmarks for evaluating the operational costs of database implementations. The benchmarks were developed using AP-203, but can be applied to any of the STEP models.

- Measurements of SDAI binding operational characteristics on database systems that are commonly used by engineering applications.

- Recommendations for implementors based on capabilities and the relative costs of implementation and operation for each database and implementation architecture.

These contributions should simplify engineering database construction by providing a well-defined framework, examples, and recommendations. In addition, this work may improve the quality of implementations by ensuring design decisions appropriate to the intended use of the system. In the larger view, it is hoped that these contributions will help industry to integrate design and manufacturing processes and reap the benefits of concurrent engineering.

1.4 Thesis Organization

Chapter Two is a historical review of STEP, EXPRESS, and information modeling.

Chapter Three describes the framework for SDAI database implementations. We examine the major tasks of an implementation project and investigate design decisions facing an implementor. This chapter also reviews earlier work as it pertains to each stage of the implementation process.

Chapter Four estimates the implementation costs for each of the implementation techniques described in the previous chapter by surveying working form SDAI, upload download, cached and direct SDAI implementations built on a variety of databases.

Chapter Five describes a set of benchmarks to evaluate the operational costs of implementations. We discuss the AP-203 information model and each of the benchmarks.

Chapter Six describes the equipment and data sets used for the benchmark measurements, and presents the measurement results. Each benchmark was run on the test implementations under a variety of conditions and with a number of optimizations.

Chapter Seven discusses the measurements and looks at the effect of various factors, such as implementation architecture, cost per access, and various optimizations, as alternatives to SDAI operations.

Chapter Eight summarizes the conclusions and contributions of this work and discusses areas of future interest.

Chapter Nine lists references to the literature cited in this document.

2 Historical Review

2.1 Overview

This chapter provides background information on engineering information models. Section 2.2 presents relevant background material on the EXPRESS language and information modeling. Section 2.3 describes the STEP standard, including the organization of the STEP models and the two primary implementation methods: File Exchange and the Standard Data Access Interface (SDAI). Section 2.4 describes other notable projects that have used the EXPRESS language to develop information models. Section 2.5 discusses other relevant standards groups, such as OMG and ODMG.

2.2 Information Modeling and the EXPRESS Language

This section presents some background material on EXPRESS and information modeling. For a complete treatment of the subject, the reader is encouraged to refer to Schenck and Wilson [Sche94] or the EXPRESS reference manual [ISO94b].

2.2.1 Information Modeling

Raw data is not information. Two parties can only exchange data in conjunction with an agreement on the meaning of the data. Consider the number “1964.” This number is data without information. The data becomes useful if we add the information that it is a year (1964), or the number of tissues used during an average head cold (1964). Although the data is the same in both cases, the information is different.

An information model is an agreement on the meaning of data. Early CAD standards, such as IGES [IGES80], usually focused on data exchange without a formal description of the underlying information model. EXPRESS has been designed to represent these information models in a formal manner.

An information model addresses the underlying meaning of data regardless of technology. A model describes meaning through structure and correctness constraints. It does not specify encoding techniques for data values. When two parties agree upon an information model, they can map the model into a particular exchange technology. For example, if two applications shared an information model for years, they might transmit the data describing the year 1964 as the 32-bit integer “0x000007AC,” as the IEEE 764 single precision floating point number “0x44F58000,” or as the ASCII string “0x31 0x39 0x36 0x34.” Each of these examples uses a different data exchange technology, but they all correspond to the same information model.

The EXPRESS language is used to describe technology independent information models. Because of this, issues like representation precision and execution speed are not considered as modeling issues. As we will see, these concerns will only be addressed when an information model is eventually mapped into the data model of an underlying storage mechanism.

2.2.2 Why EXPRESS is Important

EXPRESS is a formal language for specifying information requirements. EXPRESS is an ISO standard (ISO 10303-11 [ISO94b]) and has been used by STEP, POSC, DICE, CFI, and other projects to describe the information requirements of many engineering activities. EXPRESS has several strengths:

- The language may be used to describe constraints as well as data structures and relationships. These constraints form an explicit correctness standard for an information model.
- EXPRESS models are computer processable, so software may take advantage of the definitions without human transcription.
- EXPRESS has undergone the international standardization process, which represents significant consensus that the language meets the needs of industry.

2.2.3 History of EXPRESS

The history of EXPRESS begins in 1982. The Product Data Definition Interface (PDDI) project was formed in 1982 to specify an interface between design and manufacturing for product definitions [Wils87]. During this project, Douglas Schenck at McDonnell Douglas developed a data definition language called DSL [Sche94]. This language was the basis for EXPRESS.

In December of 1983, the International Standards Committee began work on the Standard for the Exchange of Product Model Data (STEP). This new standard was to define an integrated product information model. At the time, IDEF1X [IDEF85, Loom87, Bruc92], NIAM [Nijs82, Nijs89], and Entity-Relationship [Chen76] diagrams were in wide use for modeling. Lexical modeling languages such as SQL [Date89], DAPLEX [Ship81], and GEM [Zani83] were also available.

IDEF1X and NIAM diagrams were popular with the early STEP modelers, but ultimately proved unacceptable for several reasons. Neither language was an international standard and could not be normatively referenced by the STEP standard. It was seen as difficult to drive either IDEF1X or NIAM through the standardization process because other groups had ownership of them.

Furthermore, IDEF1X and NIAM were diagrammatic in nature and there was a need for a lexical form which would be easy to read, write and process. In response to this need, Doug Schenck and Bernd Wenzel introduced a prototype of the EXPRESS language to the STEP effort in 1986.

The language went through many revisions, and many concepts for structure and constraints were experimented with and refined. Development was done concurrently with the use of the language by the STEP modelers, so feedback was rapid and focused. EXPRESS is a very pragmatic language because of these influences.

Throughout these many revisions, EXPRESS acquired design concepts from Ada, Algol, C, C++, Euler, Modula-2, Pascal, PL/I, and SQL. The language developed an object-oriented flavor, with objects, inheritance, and a rich collection of types.

In the years since the early days of STEP, other modeling techniques have appeared. There are extensions to the Entity Relationship model, such as the Entity Category Model [Lars89], the Two Stage ER [Hsu89], and the Enhanced ER Model [Elma89]. Techniques such as OMT [Rumb91], Booch, and Shlaer-Mellor have also come into use. These notations are covered in detail by Schenck and Wilson [Sche94, Wils91] and by Hull and King [Hull87]. Recently, the Unified Modeling Language (UML) was developed by Booch, Jacobson, and Rumbaugh to blend OMT diagrams, Booch diagrams, and other visual modeling techniques [Booc98]. Many of these techniques enjoy popularity, and recent work by Sanderson has shown transforms from many of these notations to EXPRESS [Sand93].

2.2.4 Language Concepts

The function of EXPRESS is to describe information requirements and correctness conditions necessary for meaningful data exchange. For example, an EXPRESS information model might describe a balanced binary tree as binary tree structures with constraints that must be met by an instance such that it is balanced.

EXPRESS is not an implementation language like C++ nor a functional interface description language like CORBA/IDL. Our example model does not need to describe how items are inserted into or deleted from a binary tree in order to exchange an instance of a tree.

An EXPRESS information model is organized into schemas. These schemas contain the model definitions and serve as a scoping mechanism for subdividing large information models. Within each schema are three categories of definitions:

- **Entity Definitions** — Entity definitions describe classes of real-world objects with associated properties. The properties are called attributes and can be simple values, such as “name” or “weight,” or relationships between instances, such as “owner” or “part of.” Entities can also be organized into classification hierarchies, and inherit attributes from supertypes. The inheritance model supports single and multiple inheritance, as well as a new type, called AND/OR inheritance.

- **Type Definitions** — Type definitions describe ranges of possible values. The language provides several built-in types, and modeler can construct new types using the built-in types, generalizations of several types, and aggregates of values.

- **Correctness Rules** — A crucial component of entity and type definitions are local correctness rules. These local rules constrain relationships between entity instances or define the range of values allowed for a defined type. Global rules can also make statements about an entire information base.

- **Algorithmic Definitions** — An information modeler may also define functions and procedures to assist in the algorithmic description of constraints.

The missing features of EXPRESS are also interesting. The language does not include a construct for creating entity instances or an assignment statement for setting attribute values. Therefore, it cannot be used to create or modify a database. Furthermore, EXPRESS does not allow definition of methods, so it is not an object-oriented programming language. EXPRESS is a specification and requirements language, not a procedural language.

2.2.5 Summary

EXPRESS provides a rich collection of types and inheritance organizations to capture data structure. Entities represent real-world objects and can be organized into complex inheritance graphs. There are simple values such as reals, integers, and strings, named types to capture the meaning behind simple values, enumerations to describe a range of symbolic values, several varieties of aggregate, and unions of different types.

EXPRESS can describe complex functional and algorithmic constraints. There are ways to capture existence dependencies, keys, optional values, derived values, constraints on relationships, instances and the entire information base. The language contains a rich set of expressions, structured programming constructs, and a library of built in functions.

2.3 Standard for the Exchange of Product Model Data (STEP)

The Standard for Exchange of Product Model Data (STEP) defines specifications for the representation and exchange of digital product information. STEP was born in December of 1983, when the International Standards Organization (ISO) formed the TC184/SC4 committee. This group was to define a product data standard that incorporated experience from national efforts such as IGES [IGES80], VDAFS [VDAF86], SET [SET85], ESPRIT CAD*I [Kros89], and PDDI [Birc85, PDDI84]. The origins of the STEP effort are documented in [Wils93], and a history of the early national efforts can be found in [Wils87]. At the time of this writing (1998), the STEP effort is still very active. Many portions of STEP have been published as international standards, but many more are still under development.

2.3.1 Structure of STEP

Digital product data must contain enough information to cover a product's entire life cycle, from design to analysis, manufacture, quality control testing, inspection, and product support functions. In order to do this, STEP must cover geometry, topology, tolerances, relationships, attributes, assemblies, configuration and more.

To accomplish this ambitious goal, STEP has been divided into a multi-part standard. The STEP parts cover general areas, such testing procedures, file formats, and programming interfaces, as well as industry-specific information. STEP is extendable. Industry experts use EXPRESS to detail the exact set of information required to describe products of that industry. These *Application Protocols* form the bulk of the standard, and are the basis for STEP product data exchange.

Figure 2.1 shows the parts of the STEP Standard. The infrastructure parts, such as the Description Methods (EXPRESS) and Implementation Methods (file and programming interface), have been separated from the industry-specific parts (application protocols). Most of the infrastructure is complete, but the industry-specific parts are open-ended. Application protocols are available for mechanical and electrical products, and are under construction for composite materials, sheet metal dies, automotive design and manufacturing, shipbuilding, the AEC industry, process plants, and others. Over time, many industries will develop their own application protocols.

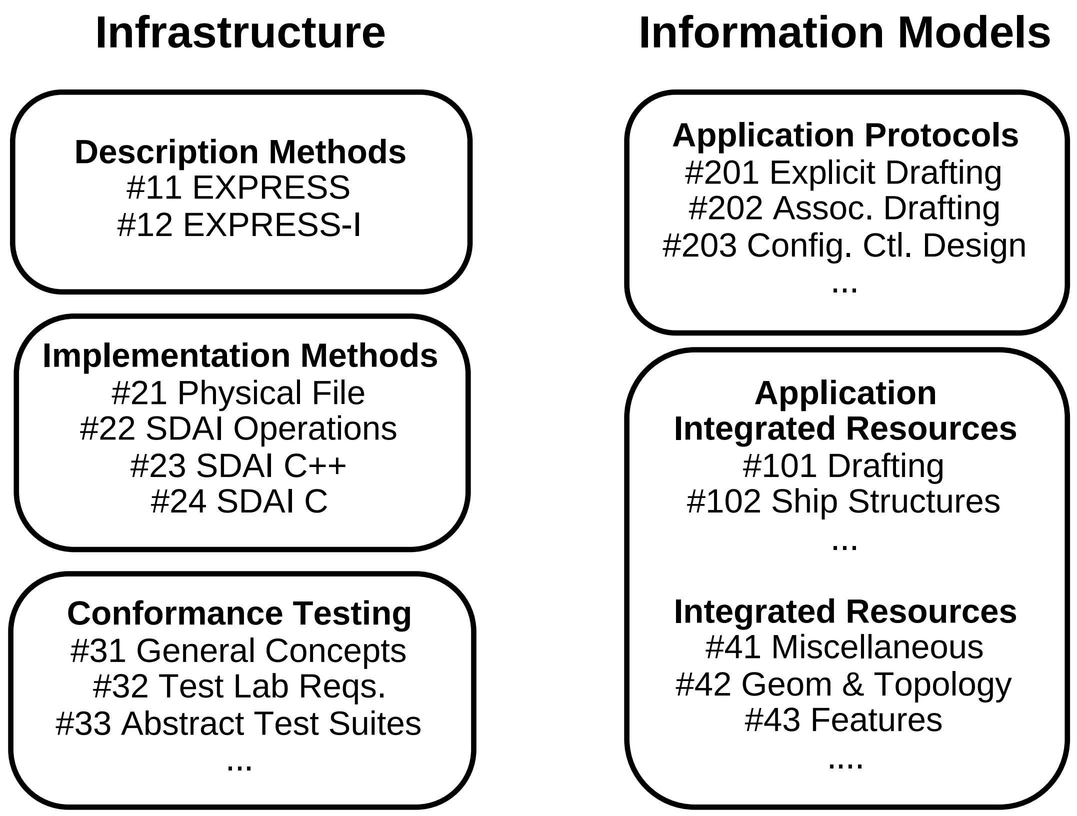

Figure 2.1 — High Level Structure of STEP

2.3.2 STEP Information Models

STEP is based on information models. These models concentrate the standardization efforts on information content, rather than implementation technology. This insures that efforts involved in developing the standard will not be discarded upon a change in computing technology. There are three classes of STEP information models:

- Application Protocols (APs).
- Integrated Resources (IRs).
- Application Integrated Constructs (AICs).

The Application Protocols are industry-specific information models for exchanging data about activities in the life cycle of a product. These protocols are built from general information models called Integrated Resources. In addition, STEP defines collections of common definitions that can be shared between Application Protocols. These Application Integrated Constructs are important when using data defined by several APs.

Application Protocols (APs)

The STEP standard defines an open-ended number of Application Protocols (APs) for industry-specific product data exchange. These APs are formal documents that cover a set of activities in the life cycle of a product. Every AP defines a set of activities, information requirements within this scope, and a formal schema for these requirements that is tied to an integrated product model shared between all APs.

The STEP application protocols are designated as the 200-series documents. A list of current APs are shown below. Some of these have reached international standard status while others are still under development.

Part 201	Explicit Draughting
Part 202	Associative Draughting
Part 203	Configuration Controlled Design
Part 204	Mechanical Design Using Boundary Representation
Part 205	Mechanical Design Using Surface Representation
Part 206	Mechanical Design Using Wireframe Representation
Part 207	Sheet Metal Dies and Blocks
Part 208	Life Cycle Product Change Process
Part 209	Design Through Analysis of Composite and Metallic Structures
Part 210	Electronic Printed Circuit Assembly, Design and Manufacturing
Part 211	Electronics Test Diagnostics and Remanufacture
Part 212	Electrotechnical Plants
Part 213	Numerical Control Process Plans for Machined Parts
Part 214	Core Data for Automotive Mechanical Design Processes
Part 215	Ship Arrangement
Part 216	Ship Moulded Forms
Part 217	Ship Piping
Part 218	Ship Structures

Table 2.1 — STEP Application Protocols

Part 219	Dimensional Inspection Process Planning for CMMs
Part 220	Printed Circuit Assembly Manufacturing Planning
Part 221	Functional Data and Schematic Representation for Process Plans
Part 222	Design Engineering to Manufacturing for Composite Structures
Part 223	Exchange of Design and Manufacturing DPD for Composites
Part 224	Mechanical Product Definition for Process Planning
Part 225	Structural Building Elements Using Explicit Shape Rep
Part 226	Shipbuilding Mechanical Systems

Table 2.1 — STEP Application Protocols

Each AP covers a portion of a product lifecycle. For example, APs 202 through 209 handle aspects of the design and analysis of mechanical parts. AP-214 further narrows this scope to automotive parts. APs 210, 211, and 220 cover aspects of circuit board manufacture. Application protocols can also be developed outside of the standards community. For example, the European Space Agency is developing an AP for the thermal analysis of spacecraft [Koni95].

As mentioned above, each AP covers a set of activities in the life cycle of a product. The statement of this scope is called the Application Activity Model (AAM). The AAM is normally documented using IDEF0 diagrams.

The next portion of an AP describes the pieces of product information required for the activities, called the Application Reference Model (ARM). This model is concise and describes requirements in terms of basic Application Objects that a user of the AP information would be concerned with. The application objects can be described by NIAM, IDEF1X, or EXPRESS-G diagrams.

Application objects can be grouped into subject areas called Units Of Functionality. The UOFs describe a logically complete subset of information about some particular product aspect. For example, the AP for configuration-controlled designs (AP-203 [ISO94f]) contains 36 application objects, distributed among nine units of functionality. The UOFs are Authorization, Bill Of Material, Design Information, Design Activity Control, Effectivity, End Item Identification, Part Identification, Shape, and Source Control.

Within each UOF are application objects that represent the information needed to describe that product aspect. For example, the Design Activity Control UOF tracks product modifications. The application objects in this UOF are: Change Order, Change Request, Start Order, Start Request, Work Order, and Work Request.

Finally, an AP document contains a conceptual schema that describes the ARM in terms of a library of pre-existing definitions. This Application Interpreted Model (AIM) is always described with EXPRESS, and is based on the definitions from the integrated resources described in the next section. AIMs are not permitted to define new entities. They are only permitted to refine definitions already present in the integrated resources. This restriction prevents the same concepts from being modeled in different ways by different APs.

Integrated Resources (IRs)

The Integrated Resources (IRs) are the heart of STEP. These conceptual schemas describe an integrated product model for all APs. There are two types of IRs. Generic integrated resources (40-series documents) describe very general characteristics of products across all industries. The application integrated resources (100-series documents) refine the integrated resources down to the needs of a particular industry. A list of current IRs are shown below. Some of these have reached international standard status while others are still in various stages of development.

Part 41	Product Description and Support
Part 42	Geometric and Topological Representation
Part 43	Representation Structures
Part 44	Product Structure Configuration
Part 45	Materials
Part 46	Visual Presentation
Part 47	Shape Tolerances
Part 48	Form Features
Part 49	Process Structure and Properties
Part 101	Draughting Resources
Part 102	Ship Structures
Part 103	Electrical/Electronics Connectivity

Table 2.2 — STEP Integrated Resources

Part 104	Finite Element Analysis
Part 105	Kinematics

Table 2.2 — STEP Integrated Resources

The resources vary in their level of detail. For example, Part 41 [ISO94d] covers product identification. Since a product could be a camshaft or an office building, these definitions are very general and are normally refined by an AP or application integrated resource. On the other hand, Part 42 [ISO94e] describes geometry, which is well-defined out of the context of any particular application, so this part is normally used without additional refinement.

Application Integrated Constructs (AICs)

STEP recently introduced a construct for describing the interoperable segments of definitions shared by multiple APs. The constructs, called Application Integrated Constructs (AICs), are sets of refined definitions that must be used as a single unit, without any additional refinements.

2.3.3 STEP Physical File Exchange

STEP defines a number of implementation methods for exchanging and manipulating information described by application protocols. The first implementation method to be defined was a straightforward ASCII file format for exchanging EXPRESS-defined data sets. This exchange file format is Part 21 of the standard [ISO94c]. A STEP exchange file contains a header section with identifying information, as well as a data section, which contains the information to be transferred. The skeleton of a STEP file is shown below:

```
ISO-10303-21;                       /* opening keyword */
HEADER;                             /* header section */
[ ... header information ... ]
ENDSEC;

DATA;                               /* data section */
```

```
[ ... entity instances ... ]
ENDSEC;
END-ISO-10303-21;                   /* closing keyword */
```

The **HEADER** section information from a Part 21 file describes identifying information about the file. An example header is shown below:

```
HEADER;
FILE_DESCRIPTION (
   ('Sample NURBS geometry for a Boeing 707',    /* description */
    'for the Common STEP Tasks tutorial'),
    '1');                                        /* impl level */
FILE_NAME (
    'ap203_database',                            /* name */
    '1995-05-18T14:18:59-04:00',                 /* timestamp */
    ('Blair Downie'),                            /* author */
    ('STEP Tools Inc.',                          /* organization */
     'Rensselaer Technology Park',
     'Troy, New York 12180',
     'info@steptools.com'),
    'ST-DEVELOPER v1.4',                         /* preprocessor */
    '',                                          /* originating system */
    '');                                         /* authorization */
FILE_SCHEMA (('CONFIG_CONTROL_DESIGN'));         /* schema */
ENDSEC;
```

The **DATA** section of a file contains entity instances. Each instance has an integer identifier. These **#nnn** numbers are used to refer to objects within the file. These numbers are unique within a file, but need not be preserved over time.

Entity instances are normally written using an “internal mapping” where the name of the entity type is followed by a list of attributes in superclass-to-subclass order. The following are some entities written using the “internal mapping.”

```
#57=DATE_AND_TIME(#58,#59);
#58=CALENDAR_DATE(1993,17,7);
#59=LOCAL_TIME(13,47,28.0,#29);
```

EXPRESS AND/OR complex entities instances have more than one type, so they are be written to a STEP file using a different encoding, called an “external mapping.” This technique encodes an object as a list of individual types, where each type contains only the attributes defined by that type. The example below shows the entry for an object that is both a B-Spline Surface with Knots and a Rational B-Spline Surface:

```
#10 = (
   BOUNDED_SURFACE ()
   B_SPLINE_SURFACE (3, 3, ((#20, #60, #100, #140),
        (#30, #70, #110, #150), (#40, #80, #120, #160),
        (#50, #90, #130, #170)), $, .F., .F., .F.)
   B_SPLINE_SURFACE_WITH_KNOTS ($, $, (0., 0., 0., 0., 1., 1., 1., 1.),
        (0., 0., 0., 0., 1., 1., 1., 1.), $)
   GEOMETRIC_REPRESENTATION_ITEM ()
   RATIONAL_B_SPLINE_SURFACE (((1., 1., 1., 1.), (1., 1., 1., 1.),
        (1., 1., 1.,1.), (1., 1., 1., 1.)))
   REPRESENTATION_ITEM ()
   SURFACE () );
```

All of the supertypes are present, even the ones that have no attributes.

STEP physical files are tightly bound to the EXPRESS schema they were written against. Because the ordering of attribute values is determined from the EXPRESS schema, changes to the schema may cause problems with files written against the original version.

2.3.4 STEP Data Access Interface (SDAI)

The second STEP implementation method is an access protocol for EXPRESS-defined databases, called the Standard Data Access Interface (SDAI). The goal of this protocol is to reduce the cost of integrated product databases by making complex engineering applications portable across database implementations.

The SDAI is described by several ISO standards documents. STEP Part 22 [ISO95a] contains a functional description of the SDAI operations, while Parts 23 [ISO95b] and 24 [ISO95c] describe how these operations are made available in the C++ and C language environments. Bindings for CORBA/IDL and Java are also being considered. All of these documents are currently under development.

In addition to operations, SDAI applications can access an EXPRESS-defined session model — an internal data set that describes the state of the SDAI session. The session model is created and modified as side effects of various SDAI operations. It keeps track of open data sets, transactions, access modes, error logs and so forth. Some SDAI bindings also provide a dictionary of EXPRESS definitions. The form of this data dictionary model is itself described using EXPRESS.

The SDAI also describes a logical database organization consisting of repositories, models, and schema instances. Each *repository* represents physical data storage, such as a file or relational database. An *SDAI model* is a named cluster of entity instances. A model is stored within one repository. A *schema instance* is a collection of many models, possibly from different repositories, that acts as the boundary for global rule checking and inter-model references.

In general, the SDAI language bindings can be classified into two groups, early and late binding, depending on whether the EXPRESS data dictionary is available to the software environment. An early binding has no data dictionary, while a late binding makes the EXPRESS definition for each object available to an application at run-time.

SDAI Early Bindings

An early binding system creates specific programming language data structures for each definition in an EXPRESS model. For example, an early binding, such as the SDAI C++, contains specific classes for each definition in AP-203. An advantage to this approach is that the C++ compiler can perform extensive type checking on an application. Special semantics or operations may also be captured as operations tied to a particular data structure.

The classes for an early binding are normally generated by an EXPRESS compiler. Entities are converted to classes, types are converted to either classes or typedefs, and the EX-

PRESS inheritance structure is mapped onto the C++ classes. Each class has access and update methods for the stored attributes, and constructors to initialize new instances. Below is a simple EXPRESS definition that would be translated to an SDAI C++ class:

```
ENTITY Point;
    x : REAL;
    y : REAL;
END_ENTITY;
```

An application can use class methods to create instances, populate them, and write them to a STEP repository. An application can also open a repository and view the contents as instances of these classes. The example SDAI C++ code below creates a **Point** object and fills in some of its attributes:

```
/* Create a point using the default constructor
 *  and use the update methods to set its values. */
SdaiModelH mod;
PointH point1 = SdaiCreate(mod,Point);
point1->x (1.0);
point1->y (0.0);
```

The object is created using a special version of the "new" operator and the attribute values are set using generated member functions.

SDAI Late Bindings

An SDAI late binding, such as the SDAI C, uses an EXPRESS data dictionary for access to data values. Generated data structures are not used. Only one data structure is used for all of the definitions in an EXPRESS model.

Data values are found by queries against the data dictionary. Applications use a few simple functions to get and retrieve values by attribute name rather than by using specialized functions for each value. The example SDAI C code below creates a **Point** object and fills in some of its attributes:

```
/* create new instances */
```

```
SdaiAppInstance point1;
point1 = sdaiCreateInstanceBN (myModel, "Point");
sdaiPutAttrBN (point1, "X", sdaiREAL, 1.0);
sdaiPutAttrBN (point1, "Y", sdaiREAL, 0.0);
```

In general, late bindings are useful in programming environments that do not use strong type checking, and in software that works on data from a common subset of multiple EXPRESS schemas. The EXPRESS model in the dictionary may change somewhat without necessarily affecting an application.

Another advantage is simplicity. An average AIM may contain upwards of 200 definitions, each of which would become classes that must be generated, compiled and linked into an early-bound application. A data dictionary-driven binding requires less initial work and lends itself to faster prototyping, although the lack of compile-time type checking is a disadvantage that will surface in larger systems.

2.3.5 STEP Summary

Some key reasons why STEP is important:

- STEP is a standard that can grow. It is based on a language (EXPRESS) and can be extended to any industry. A standard that grows will not be outdated as soon as it is published.

- STEP product models contain EXPRESS constraints as well as data structure. Formal correctness rules will prevent conflicting interpretations. STEP CASE tools use these descriptions to create more robust, maintainable systems.

- STEP is international, and was developed by users, not vendors. User-driven standards are results-oriented, while vendor-driven standards are technology-oriented. STEP has, and will continue to, survive changes in technology and can be used for long-term archiving of product data.

The reader is encouraged to consult [Owen93] or [ISO94a] for more discussion about the contents of the STEP standard, and the use of EXPRESS information models within the standard.

2.4 Other Engineering Initiatives Using EXPRESS

The following engineering initiatives have also built EXPRESS information models. These models do not use the same integrated library of definitions as the STEP models, but could still be implemented using the techniques described in this thesis.

2.4.1 Petrotechnical Open Software Corporation (POSC)

The Petrotechnical Open Software Corporation (POSC) is a consortium of Petroleum companies that has been established to define, develop, and deliver, an open systems Software Integration Platform (SIP) for use across the petroleum exploration and production industry. This SIP will be a set of standards for vendor software and is expected to improve system integration, reduce training time, and reduce software costs. The SIP will contain a set of comprehensive information models, specific exchange file formats, and user interface "look and feel" specifications that software developers can use to improve interoperability [POSC92a].

POSC and STEP have similar product data exchange goals, and POSC has adopted EXPRESS as their information modeling language [POSC92b]. STEP is defining application protocols for many industries and products, while POSC is concentrating on one industry. It is reasonable to expect the POSC information models to eventually be incorporated into STEP as Application Protocols.

The POSC organization has defined a file format called RP-66 for the exchange of seismic data. This format is based upon a fixed information model but has a performance edge in some situations. This specialization may make it difficult for vendors to respond to the evolving needs of the user community, so POSC will also need a more general purpose exchange format, such as the STEP Part 21 physical file format.

2.4.2 CAD Framework Initiative (CFI)

The CAD Framework Initiative (CFI) is a consortium of companies that has been established to define a means for CAD tools to create, access, or modify electronic design data in other tools or databases. The initial scope of the project is limited to hierarchical electrical connectivity information for tools such as logic simulators, timing, analyzers, layout tools, and so forth [CFI92].

CFI has chosen to define their connectivity information model using EXPRESS. On top of this model, they have layered a programming interface called the Design Representation Programmers Interface (DR PI). The DR PI operations understand the semantics of the data to a greater degree than that in the EXPRESS, so that DR PI methods can be used to manipulate and change a circuit database during a design session.

2.4.3 DARPA Initiative in Concurrent Engineering (DICE)

In traditional design, one engineer works on a design at a time. As enterprises grow larger, engineers become more specialized and concurrent engineering becomes more practical [Winn88]. The DARPA Initiative in Concurrent Engineering (DICE) project was begun in 1988 to create a support framework for handling concurrent design and manufacture of mechanical and electrical components. As a part of this project, EXPRESS was used to define a Product, Process, and Organization (PPO) information model.

2.5 Other Standards Efforts

The following two vendor organizations are producing technology standards that may be of some interest in the context of engineering databases and EXPRESS.

2.5.1 Object Management Group (OMG)

The Object Management Group (OMG) is a software industry consortium developing specifications for large-scale distributed applications (open distributed processing) using object-oriented methodology [OMG93].

The OMG was founded in April 1989, and is composed of large and small vendors (IBM, Canon, DEC, Philips, Olivetti, AT&T, Sun Microsystems, Informix, ICL, Enfin Systems, Architecture Projects Management, Apple Computer, O2 Technology, etc.) as well as end-user companies (Citicorp, American Airlines, British Telecom, John Deere, etc.)

Work in the consortium originally focused on specifications for an Object Request Broker (ORB), for handling distribution of messages between application objects. In late 1991, several members of the OMG consortium proposed the Common Object Request Broker Architecture (CORBA). This specification adds an object-oriented interface to the Remote Procedure Call (RPC) mechanism. This makes it possible for object-oriented applications to dynamically call each other's methods at run time. Dynamic calling allows a service to be replaced at run time without affecting the operation of other applications.

The first phase of CORBA described the basic request broker architecture and gave applications an Interface Description Language (IDL) that can describe their services to other applications. The next phase of CORBA is seeking to define a set of standard services for all applications. One of these services is persistence. Other services include change management, version control, relationship management, and data interchange.

If the OMG effort is successful, then database systems will be required to divide their services so that applications only have to pay for the ones that they need. However, the degree to which this can be done is controversial. Today there are few, if any, examples of systems that have divided their services in the way detailed by the OMG.

2.5.2 Object Database Management Group (ODMG)

The Object Database Management Group (ODMG) is a working group of ODBMS vendors who are cooperating to further refine the OMG specifications into an interface that all vendors can support, allowing application portability and interoperability [ODMG93]. They are working on an object definition language (ODL) which is an extension to the OMG IDL and an object query language (OQL) which provides declarative access for queries. The OQL language is an extension of SQL.

The ODBMS vendors can be distinguished by the degree to which they are willing to trade performance for functionality. The high performance databases only provide persistence. They do so by intercepting memory faults, so these databases require minimal changes to existing programs. The high function databases require many changes to applications, but in return they extend object definitions to provide a greater range of services such as version control, relationship management, constraint execution and so on.

This tension has split the ODMG members into two camps. The functionality camp has more members (and therefore has more votes in ODMG) but its market size is smaller. Database vendors in the functionality camp include Objectivity and Versant. Object Design (ObjectStore) belongs to the speed camp and opposes efforts to add functionality requirements to the ODMG specifications.

3 Framework for EXPRESS Database Implementations

3.1 Overview

EXPRESS information models describe logical structures that must be mapped to a software technology before they can be used. Section 3.2 reviews the four STEP implementation levels. We focus on the third level — database technology. The remaining sections study implementation decisions and propose a framework of SDAI implementation architectures based on how they handle data access.

3.2 STEP Implementation Levels

EXPRESS information models describe logical structures. These structures must be mapped to a software technology before they can be used. Consider the example in Section 2.2.1. An information model may describe the notion of “years,” but this description must be mapped into a specific technology — such as an ASCII string, 32-bit integer, or IEEE 764 number — before it can be used.

This technology independence is a strength when developing information models that could be used for a long time. A model might be implemented using many data processing technologies and would remain relevant as new technologies appear. Four levels of implementation technology have been identified for EXPRESS models [Wils90]. These levels are shown in Figure 3.1.

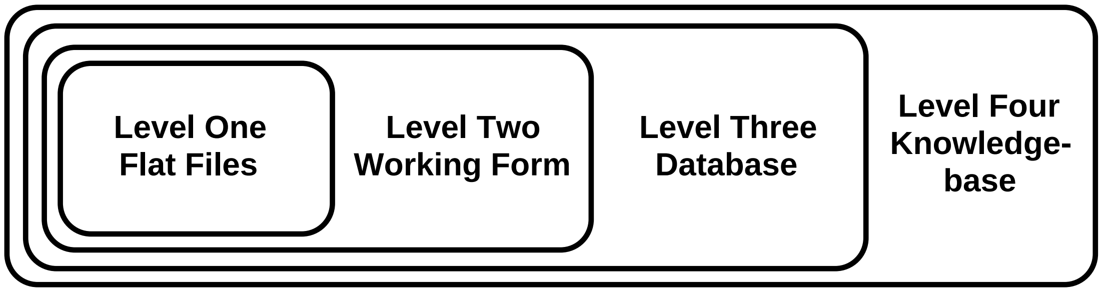

Figure 3.1 — STEP Implementation Levels

3.2.1 Level One — File Exchange

A Level One implementation is the least complex. At this level, EXPRESS-defined product data is passed between applications using flat files. The STEP Part 21 format has been defined for this purpose, although other encoding specifications could be used.

An application must simply read and write files. It does not need any other features. In particular, there are no constraints upon the representation of product data within the application. It may read the EXPRESS-defined data file using a dedicated parser and immediately convert the instance data into some other structure. The application does not need to use the EXPRESS model for operating on data, only for reading and writing files.

Level One implementations were the first to appear and are now quite common. Many CAD vendors have built Part 21 interfaces that support the exchange of AP-203 data. Some PDM vendors are also beginning to add file exchange interfaces to their systems.

3.2.2 Level Two —þWorking-Form

A Level Two implementation has all features required by Level One, plus the ability to manipulate data based upon the EXPRESS information model. A working-form application communicates with other applications using exchange files. However, when an application reads a file into memory, the data is made available to the code in a form organized and described by the EXPRESS model.

The SDAI has been developed as a standard API for working-form applications. The SDAI functions allow programs to manipulate any product data defined by an EXPRESS model. Other programming bindings could also be used, as long as they are based upon EXPRESS models.

A number of working-form implementations have been built, such as the ST-Developer ROSE and SDAI libraries [Hard91] and the NIST SCL [Clar90,Saud95]. Working-form bindings are often used just for their Level One file exchange properties. In fact, most of the CAD system Part 21 interfaces have been built around working-form bindings.

3.2.3 Level Three — Database

A Level Three implementation has all features required by Level Two, plus the ability to work with data stored in a Database Management System (DBMS). Databases organize large quantities of information [Gall84] and an integrated product database may store data that covers many aspects of the engineering life cycle. Multiple applications can access the product data, and may take advantage of database features such as query processing.

The implementation should also be able to read and write exchange files and make product data available using either the SDAI or another API that presents data as EXPRESS-defined structures. A database implementation may support validation of some EXPRESS constraints, but need not support them all.

Limited work has been done on SDAI database implementations. As we will discuss in Section 3.4, several groups have looked at mapping EXPRESS-definitions into database structures, but few have attempted to provide SDAI access to the structures. The first SDAI database prototype was based on Objectivity [Whit91]. Herbst describes some work with ObjectStore [Herb94] and Erlangen University has investigated a number of systems [Kreb95a]. The following chapter describes SDAI systems constructed at Rensselaer and STEP Tools using the ObjectStore, OpenODB, Oracle, and Versant database systems.

3.2.4 Level Four — Knowledgebase

A Level Four implementation has the features of all lower implementations, as well as full support for EXPRESS constraint validation. A knowledgebase system should read and write exchange files, make product data available to applications as EXPRESS-defined structures, work on data stored in a central database, and should be able to reason about the contents of the database.

Knowledgebase systems encode rules using techniques such as frames, semantic nets, and various logic systems, and then use inference techniques such as forward and backward chaining to reason about the contents of a database. Consult [Brac85] for more information on knowledge representation systems.

Knowledgebase implementations do not exist, although some interesting preliminary work was done by the PreAmp project [Gadi94, Mull93]. The PreAmp project built an AP-210-based system for analyzing the manufacturability of electronic circuit boards. The key component of this was a “manufacturability advisor,” which loads AP-210 data into an Intellicorp Kappa database, where rules analyze aspects of the design.

3.2.5 Summary

EXPRESS information models describe logical structures that must be mapped to an implementation technology before use. The file and working-form implementations have the fewest requirements and are widely available. Database and knowledgebase implementations have more requirements and are not common. The following sections focus on ways to satisfy the requirements for database (Level Three) implementations.

3.3 Database Implementation Process

In order to construct an engineering database around an EXPRESS information model, we must:

- Define the database structures from EXPRESS.
- Provide SDAI access to the database.

The first task has been well-researched and is summarized in Section 3.4.

The second task requires several design decisions. An implementor must decide how to transfer data between database and application. Section 3.5 identifies three architectures: file upload and download with working-form SDAI, cached SDAI, and direct SDAI. Other decisions are discussed in Section 3.6, such as how to make EXPRESS structure definitions available and how constraints might be validated.

3.4 EXPRESS to Database Schema

Implementors must convert an EXPRESS information model into schema definitions for the target database. This conversion requires a mapping from the EXPRESS language into the data model (DDL) of the target database system.

The literature contains mappings from EXPRESS to many of the popular data models. For example, McDonnell/Douglas [Egge88], Rutherford Appleton Laboratory [Mead89], NIST [Morr90], and Rensselaer [Ragh92] have shown mappings of EXPRESS to the relational model. Erlangen University has shown a mapping to the Postgres extended relational model [Kreb95b].

Sanderson and Spooner describe mappings between EXPRESS and the network, hierarchical, and relational models. This work also shows that EXPRESS is semantically richer than these models [Sand93]. Sanderson has also shown a general mechanism for identifying information loss between data models [Sand95].

Some object-oriented database systems use programming languages as a DDL. Work on C++ mappings has been done by the Norwegian Institute of Technology in Trondheim [Totl92], STEP Tools [STI92c], and NIST [Clar90]. Additional work on programming language mappings has been done by STEP WG11 during the development of the SDAI bindings [ISO95b, ISO95c].

An implementor can use an existing mapping if applicable, but some database systems may require a new EXPRESS to DDL mapping. EXPRESS information models can challenge the capabilities of existing database systems. In particular, the following features of EXPRESS may require encoding or other manipulations to preserve the original information within the native data model:

- **Entities** — Entity instances do not require unique keys formed from attribute values. Instead, each instance has an implicit identifier, which is used to store relationships. Systems that rely on unique keys for identification will probably need to add additional identifier data to entities.

- **Inheritance** — The EXPRESS inheritance model is rich. It includes single inheritance and multiple inheritance. It also supports AND/OR inheritance, where an instance may have a set of types. Since inheritance implies duplication of attributes between supertypes and subtypes, normalization [Date86] may be needed for some database systems.

- **Primitive Types** — EXPRESS supports seven primitive types — integer, real, number, string, binary, boolean, and logical. New types can be defined by adding constraints to existing types. Encoding may be needed for some of the primitive types or defined types.

- **Enumerations** — Each set of enumerated values is in a separate name space. Not all database systems support enumerations as a primitive type. For example, these might require simulation as a foreign key to a separate table of enumerators.

- **Selects** — The EXPRESS select type is analogous to a strongly typed union and is used to group disjoint types. Selects can be formed from any number of base types, and can be nested to arbitrary depth. Few database systems support a union type. It may be necessary to simulate this structure using a vector with discriminant or other technique.

- **Aggregates** — EXPRESS supports ordered and unordered aggregates formed of any base type and nested to arbitrary depth. These types of structures are only supported by non-first normal form databases. Even with these, some aggregate styles may need to be simulated.

A mapping from EXPRESS to a database schema should address each of these constructs. Depending upon the database, an implementor may also be able to address derived attributes, local and global constraints, unique, or inverse clauses.

3.5 SDAI Access Architectures

SDAI binding software uses the database DML to transfer EXPRESS-defined data between database and application. The most important design decision facing an implementor is how to transfer data between database and application. Based on the quantity of data and time of transfer, we identify three architectures:

- Entire model, off-line batch transfer — File Upload/Download SDAI Binding.
- Entire model, on-line batch transfer — Cached SDAI Binding.
- Individual values, on-line transfer — Direct SDAI Binding.

We discuss the characteristics of each architecture in the following sections.

3.5.1 Upload/Download Access

A file upload/download SDAI binding operates on an entire SDAI model at one time. The size and composition of the model is not rigidly determined, but it should contain all data needed for a single application run. When SDAI access is desired, the model is extracted from the database and written to a file, usually in Part 21 format. The file is read into main-memory and manipulated by an application built around a working-form SDAI binding. When the application is finished, the updated file can be loaded back into the database.

A file upload/download binding is composed of two pieces. The first is a transfer program that moves EXPRESS-defined data between the database and exchange file. The second is a working-form SDAI binding that can read an entire Part 21 data set into memory, manipulate it, and write it back out to a file when finished. Since working-form bindings operate on main memory data, they can offer extremely fast performance. Figure 3.2 shows the structure of an upload/download interface.

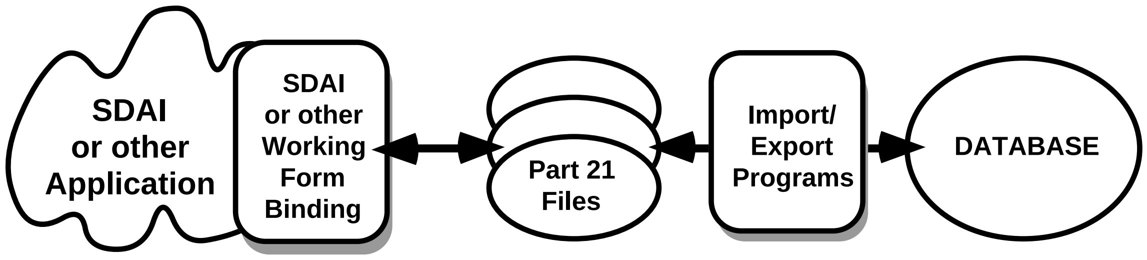

Figure 3.2 — Upload/Download SDAI Binding Structure

This SDAI architecture does not take advantage of many database features. Applications written using the native database interface could use queries, locking, and such, but SDAI applications can only use the database as a form of file system.

When the SDAI model is extracted, the database can be locked on a per-model basis to prevent conflicting updates, but locking on a smaller level would not be possible. Concurrent update on a data set would be difficult, since copies of the data would be separated by time and space once they have been extracted. Delta scripts have been proposed as way to implement concurrent update in such a situation [Hard93, Hard95a].

This architecture has a high latency since an entire model must be extracted from the database before an SDAI application can begin work. However, the extract process can be performed before the data is needed, which may be of use if the process is particularly time consuming.

The load and extract programs should only require O(N) database accesses. These programs must touch every instance in an SDAI model, perhaps through a two pass algorithm that creates instances and then fills in references. Once a model has been extracted, a working-form SDAI can read the file and access data at main-memory speeds. The file could be processed by several programs. If many programs need access to static data, this technique could be used to alleviate demand on the database server.

The model only needs to be loaded back into the database if it has changed. The load program could replace the entire model or it could try to identify and replace only what has changed.

An upload/download binding should not be too difficult to construct. The upload and download programs operate in a batch fashion, so they can be coded using straightforward algorithms. For example, they could transfer data using several passes or use global information about the data set. A new working-form SDAI library could be implemented, but it is far more cost effective to reuse an existing one.

3.5.2 Cached SDAI Access

A cached SDAI binding also operates on an entire SDAI model. Unlike the file upload/download binding, the model is transferred to and from a main-memory cache. Once in the main-memory cache, the data can be manipulated by an application built around a working-form SDAI binding. When the application is finished, the cached model can be loaded back into the database. Figure 3.3 shows the structure of a cached SDAI binding interface.

A cached binding architecture shares most characteristics of a file upload/download binding. Since the binding software does not read and write an intermediate file, a cached binding will have a slightly lower latency than a file upload/download binding.

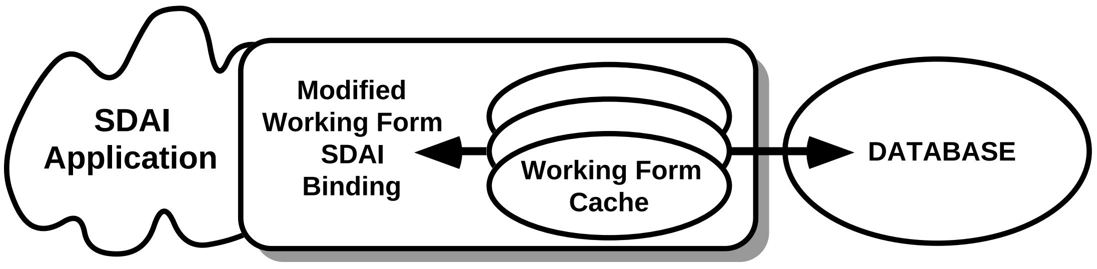

Figure 3.3 — Cached SDAI Binding Structure

If the database is fast, the cache load time may be close to the file read/write times. In this case, a cached binding would be faster and more much convenient than a file upload/ download binding.

If the database is slow and the cache load time is much larger than the file read/write times, the savings would be minimal. A cached binding must extract data while the application is running and must extract a separate copy for each application. A file upload/ download binding can extract data overnight if necessary, and can amortize the cost of extraction over several applications that use the same data.

A cached SDAI binding should require only slightly more effort to implement than a file upload/download binding. The upload and download operations must be developed as functions rather than as stand-alone programs. These must be integrated into an existing working-form binding, but they can still use straightforward algorithms. It may be desirable to develop a file upload/download binding first, then evolve it into a cached binding if necessary.

It may also be possible to construct a cached binding that operates on several different types of database. The modified working-form SDAI could have upload and download libraries for several database systems. Once loaded into the memory cache, an application could simultaneously manipulate data from many different systems.

3.5.3 Direct-Binding SDAI Access

A direct SDAI binding operates on one data value at a time. Applications manipulate the database directly, with no intermediate cache. Each operation in the SDAI binding is implemented as one or more native operations directly upon the database. Figure 3.4 shows the structure of a direct SDAI binding interface.

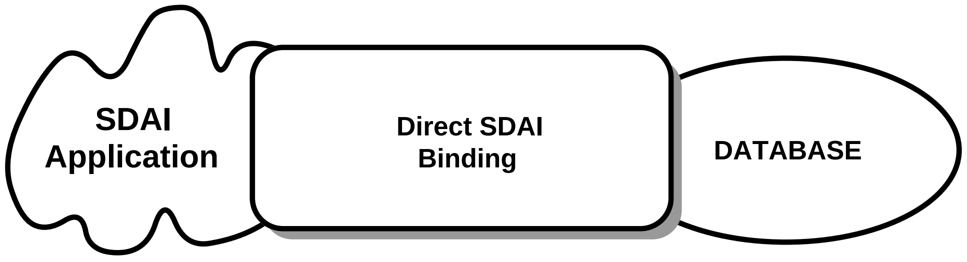

Figure 3.4 — Direct SDAI Binding Structure

This architecture has low latency since no information must be extracted from the database before an SDAI application can begin work. This SDAI architecture can take full advantage of database features. Concurrent updates and fine-grained locking can be used to the full extent supported by the database.

Previously, we noted that file upload/download and cached bindings require O(N) database operations. Generally, a direct SDAI requires a constant number of database operations for each SDAI operation. An application built with an O(log) algorithm will require O(log) database operations while an O(N^2) algorithm would require O(N^2) database calls.

A direct SDAI binding could be difficult to construct. Each operation must be implemented using native database methods. Isolated operations may require more complex algorithms than batch code, particularly if the data is heavily encoded. The binding may also need to keep state information between calls so that cursors can be opened or closed, and the mapping between SDAI object identifiers and database identifiers can be preserved.

3.5.4 Access Summary

These architectures offer a range of implementation costs and capabilities. Upload/download and cached binding systems can be built with straightforward algorithms and existing code, but can not take advantage of many database features. From the application point of view, they have a high latency, but good performance once the data is loaded.

The direct binding systems require complex algorithms and quite a bit of new code, but they can make more database features available to an SDAI binding. They offer low latency, but are more heavily influenced by the speed of the underlying database system.

The relative merits of each architecture must be evaluated in light of the database system, information model, and user requirements. Capabilities and costs increase as one progresses from an upload/download to a direct binding. The architecture must be justified against expected SDAI applications. If low latency or concurrent update is required, a direct binding must be used. If neither capability is essential, a file or cached SDAI binding could provide access at a more reasonable cost. The pros and cons of the architectures are summarized in Table 3.1.

	File Upload/ Download	**Cached**	**Direct**
Concurrent Update	No	No	Possible
Coarse-Grained (Model) Locking	Possible	Possible	Possible
Fine-Grained (Instance) Locking	No	No	Possible
Database Calls	O(N) accesses O(N) updates off-line	O(N) accesses O(N) updates on-line	As required by algorithm
Latency	High, but could be pre-fetched	High	Low

Table 3.1 — Characteristics of SDAI Access Architectures

	File Upload/ Download	Cached	Direct
System With High Operation Cost	OK, if data can be pre-fetched	Poor	Depends on amount of data accessed
System With Low Operation Cost	Good	Good (very cost effective, low implementation cost)	Good

Table 3.1 — Characteristics of SDAI Access Architectures

3.6 Other Design Considerations

Once the style of data access has been identified, an implementor must consider at least two more design factors. The first is how the database access software will be adapted to different EXPRESS structure definitions, and the second is how the software will handle EXPRESS constraints.

3.6.1 EXPRESS Binding Style

The SDAI database access software should be written so that it can be adapted for use with different EXPRESS schemas. This can be done by writing or generating specific software for each information model (early binding), or by writing general-purpose software that works from a data-dictionary representation of each information model (late binding). These approaches are illustrated in Figure 3.5.

Code Generation (Early Binding)

This approach creates custom software for every information model. An EXPRESS compiler can generate programming language data structures and functions for each definition in the model. The compiler can generate load/unload programs, or libraries of data transfer functions.

The code generated by an EXPRESS compiler can use the public database API, or undocumented features that offer performance enhancements. There is no significant penalty for using the undocumented features, since the compiler can be updated and code regenerated if features are changed or eliminated.

Many database APIs are in C, but C++ is a good language choice for generated code. C++ has strong type checking, supports inheritance, allows methods to be attached to classes, and can inter-operate with an existing C API.

Data-Dictionary (Late Binding)

This approach creates one body of general-purpose software that consults a data-dictionary for each information model. The software accesses the database through calls against the data-dictionary with names of types and attributes. This approach works well for systems that have strong data-dictionary support.

An interface that require extensive encoding of the EXPRESS model might perform better with code generation. Encoding means that the data must be assembled or disassembled when moving between interface and database. A late-bound interface must do this manipulation by interpreting the data-dictionary at run-time, while generated code can be compiled for extra speed.

EXPRESS compilers can generate programs from easily changed templates, so a generated interface may be easier to optimize than a dictionary interface. Furthermore, after being generated, programs can be changed by hand to optimize access or storage of certain entity types or attributes. The general-purpose core of a dictionary-based system must treat all structures uniformly, and makes no provisions for fine-tuning particular entities or attributes.

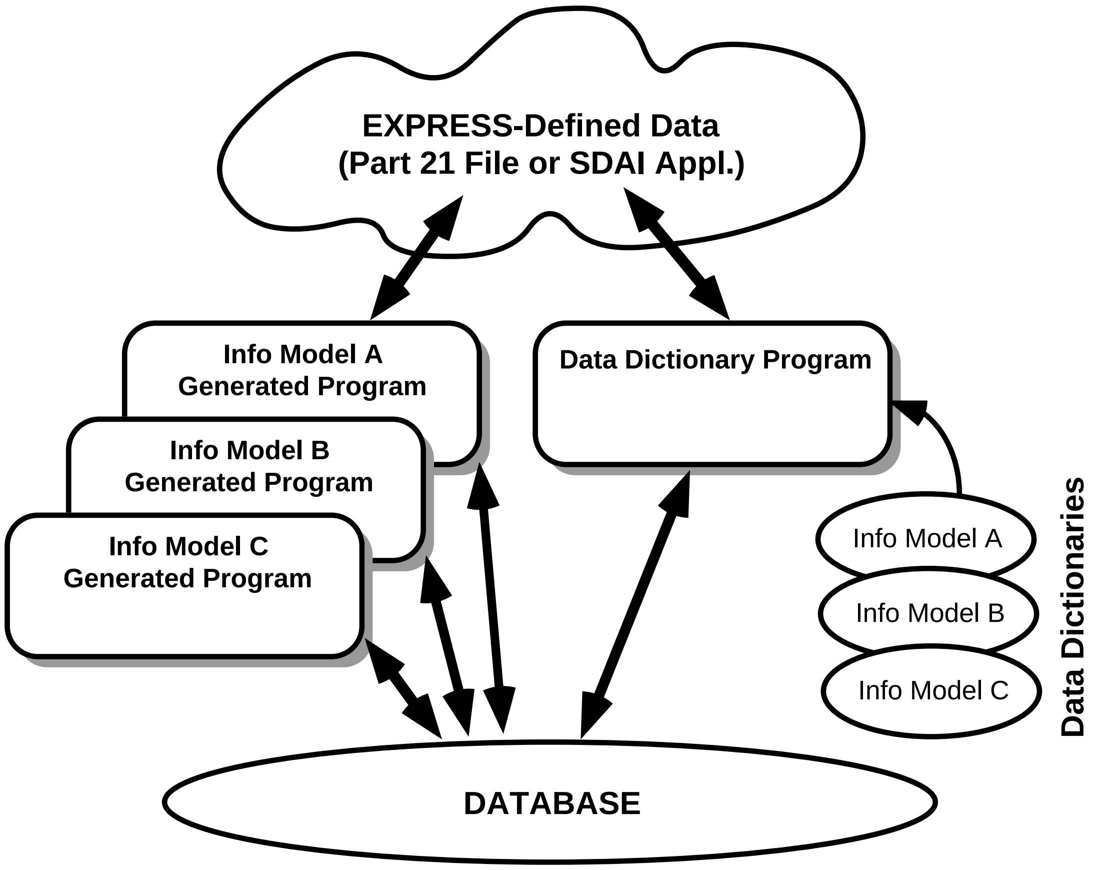

Figure 3.5 — Code Generation vs. Data-Dictionary Software

It is possible to use a mixed approach, where code generation is used for some structures and a data-dictionary for the rest. For applications that only deal with part of an information model, this approach allows for strong type checking and other benefits of code generation while reducing the number of unused definitions that must be managed. However, a database interface does not normally benefit from this technique, since an interface generally uses everything in an information model.

An implementor should consider the size of information models. Models can contain several hundred definitions. For example, the relational table definitions for AP-203 require over 1800 lines of generated SQL. This many definitions may strain a database system, and the implementor may need to adjust various parameters or add extra table space to handle the definitions.

Experience will determine whether existing database systems can even handle such large information models. Limitations in these systems may require that subsets of information models be used. Another approach that might be explored is to use an EXPRESS view mechanism to create information models that are simpler and less taxing on a particular system [Hard94].

The choice of binding style depends mostly on the database services provided by the underlying system. The characteristics of each style are summarized in Table 3.2. In situations where a system could use either, the code generation style should be given preference if customizability is important. A code generation system might also be more efficient, such as with static vs. dynamic SQL.

	Code Generation	**Data-Dictionary**
System Requirements	Any System	Run-time access through Data Dictionary
Cost to Implement	Moderate (modify EXPRESS compiler)	Moderate (new EXPRESS-driven transfer software)
Customizability	High. Code for each instance can be changed	Low. All instances are treated alike

Table 3.2 — Characteristics of Binding Styles

3.6.2 Constraint Validation

A key feature of EXPRESS is the explicit representation of constraints. Full support for constraints is feature of knowledgebase implementations, but database implementations may also provide limited support. Depending upon the database system, the software may be able to validate some of the constraints.

The different access styles can affect how and when constraints might be evaluated. Batch validation is the only option for upload/download software. Constraints can be evaluated interactively by software using a cached or direct SDAI binding. Validation might be done by code generated for each constraint or by a dictionary-based EXPRESS interpreter.

3.7 Framework Summary

Before any work can begin, an implementor must understand and be able to draw correspondence between the database data model and the range of structures representable by EXPRESS. Once a mapping has been selected or developed, work can begin on the database access software.

Reviewing the software design factors, we find a matrix of design decisions. One axis corresponds to the services provided by the software (access style), while the other corresponds to the manner in which the software is written (binding style). This breaks down to the six possibilities shown in Table 3.3

	Access Style		
Binding Style	Early-Bound Upload/Download	Early-Bound Cached SDAI	Early-Bound Direct SDAI
	Late-Bound Upload/Download	Late-Bound Cached SDAI	Late-Bound Direct SDAI

Table 3.3 — Software Design Options

If an implementation chooses to provide support for constraint validation, we have the range of approaches shown in Table 3.4.

Upload/Download	**Cached**	**Direct**
Batch Evaluation	Batch or Interactive Evaluation	Batch or Interactive Evaluation

Table 3.4 — Approaches to Constraint Validation

4 Implementation Cost Studies

4.1 Overview

In this chapter, we examine the results of several implementation projects. These projects illustrate all of the approaches to data access that were described in Section 3.5. The systems were built on a variety of databases. We look at the construction of each system and discuss the implementation costs.

Before we discuss the implementation projects, Section 4.2 examines the characteristics of each database system. Next, we look at each data access style. Section 4.3 covers file upload/download implementations built on Oracle and OpenODB. Section 4.4 discusses cached SDAI bindings on Oracle and Versant. Finally, Section 4.5 looks at direct SDAI bindings on Oracle and ObjectStore. We briefly describe the construction and EXPRESS/DDL mappings of each system. We conclude each section with a discussion of the design decisions and required implementation effort. There are many ways to measure implementation effort, but we will use lines of code to estimate relative construction costs.

4.2 Database Systems

The implementations discussed in this chapter were built using the Oracle, OpenODB, ObjectStore, and Versant database systems. In this section we discuss the characteristics of these database systems. We also discuss the working-form SDAI library used by the upload/download and cached SDAI implementations.

4.2.1 Oracle

Oracle is the most widely used of the systems examined here. Oracle and other relational systems such as DB/2 and INFORMIX are used by engineering organizations to store and manage configuration control data. The strength of relational systems is in their ability to store large amounts of data in a highly normalized, tabular form, and to perform efficient queries across large data sets. Relational systems use SQL for both data definition and data manipulation.

4.2.2 OpenODB

OpenODB, from Hewlett Packard, is a hybrid system that combines the recognized strengths of relational systems with an object data model. This system provides an object management front-end to a relational database, and introduces a new object-oriented query language, based on SQL, called OSQL. The OpenODB data model is based upon objects, types, and functions. Functions can be stored, in which case they behave as traditional attributes, or they can be computed using OSQL or external software [Open92].

4.2.3 ObjectStore

The ObjectStore object-oriented database system, from Object Design Corporation, intercepts virtual memory page faults to make C++ objects persistent without need for a special class library. The ObjectStore data model is C++ and a customized compiler is used for schema capture. Data-dictionary support and query-based access were originally limited, but have improved in recent versions of the software [Obje94].

4.2.4 Versant

Versant is an object-oriented database system from Versant Object Technology (originally Object Sciences Corporation). Versant provides a persistent C++ class library and a central server for data check in, check out, and queries. A modified C++ compiler can be used for schema capture, but the data model can be separated from C/C++ and supports programming bindings to languages like Smalltalk [Vers93].

4.2.5 ROSE

The projects in this chapter used ST-Developer, from STEP Tools, for STEP and EXPRESS development support. This package includes the ROSE C++ and SDAI C working-form libraries for application development [STI92a, STI92b]. The ROSE library predates, and has influenced, the SDAI C++ specification. It provides all required SDAI services, but the organization and naming of some function calls are slightly different.

These libraries provide Part 21 file I/O, EXPRESS data-dictionary access, programming access, and in-memory working-form cache management. The Oracle and OpenODB projects added customized code generation software to the ST-Developer EXPRESS compiler. ST-Developer also contains an EXPRESS interpreter, which was used for constraint validation.

4.3 File Upload/Download Implementations

The first two implementations are based upon Oracle and OpenODB. These implementations provide access by loading and extracting files that can be used by an SDAI working-form binding. The Oracle and OpenODB implementations are general-purpose; they may be used with any EXPRESS information model.

4.3.1 Oracle Upload / Download

The file upload/download interface to Oracle was implemented using code generation. A specially modified EXPRESS compiler generates three programs for each EXPRESS information model.

The first program defines the database schema using SQL “CREATE TABLE” statements. The remaining two programs use embedded SQL and the ROSE C++ library to move EXPRESS-defined data between a STEP Part 21 file and an Oracle database.

The upload program reads a Part 21 file into memory and makes SQL calls to create objects in the Oracle database. The download program uses SQL queries to select a data set and extract attribute values from the database. The program creates in-memory objects which are later written as a STEP Part 21 file.

EXPRESS Mapping to Oracle SQL

The Oracle implementation uses the mapping from EXPRESS to the relational model described by [Ragh92]. Each entity is mapped to a table with columns for attributes. Each table has a column with a unique identifier for each instance. Attributes with primitive values are stored in place, and composite values like entity instances, selects, and aggregates are stored as foreign keys containing the unique instance identifier.

Inheritance is normalized out of the tables. The table for each entity type contains the local attributes defined by the entity, and uses the instance identifier as the primary key. A complete entity instance, with all inherited attributes, can be reconstructed by a join on the identifier across all tables in the type hierarchy.

The Oracle primitive data types are not as extensive as those of EXPRESS. Booleans and logicals are approximated as integer values; enumerations are stored as strings; defined types of primitives are treated as the base primitive type. The corresponding EXPRESS and Oracle types are shown in Table 4.1.

EXPRESS Type	**ORACLE Type**
REAL	float/double
INTEGER	integer
BOOLEAN	integer
LOGICAL	integer
STRING	varchar
BINARY	number
ENUMERATION	varchar

Table 4.1 — Mapping from EXPRESS to the Oracle Primitive Types

The only aggregate structure that Oracle supports is a table of tuples. The EXPRESS aggregates are simulated by using a foreign key to group all elements in a particular aggregate instance. An additional index column preserves the ordering of lists and arrays.

The relational model does not directly support the union construct, so EXPRESS Selects are simulated by a table with a column for each possible member type. Only one column in each tuple contains a value. The remaining columns are null.

EXPRESS imposes no limit on the length of type or attribute names, and many information models define entities and attributes with long names. Oracle restricts the length of table and column names to 30 characters. Name length conflicts are resolved through an abbreviation algorithm.

4.3.2 OpenODB Upload / Download

The file upload/download interface to OpenODB was also implemented using code generation. A specially modified EXPRESS compiler generates three programs for each EXPRESS information model.

The first program defines the database schema using OSQL “CREATE TYPE” and “CREATE FUNCTION” statements. These statements are executed using **iosql** — the OpenODB OSQL interpreter. This defines database structures for the information model using the mappings described below.

The second and third programs use the OpenODB Oaci programming interface and the ROSE C++ library to transfer EXPRESS-defined data between a STEP Part 21 file and an OpenODB database. These programs operate in the same way as the Oracle upload and download programs.

EXPRESS Mapping to OpenODB OSQL

The OpenODB data model is object-oriented and supports object types with identity and associated stored or computed functions (attributes). EXPRESS entity types map easily to OpenODB object types. Each explicit attribute can be represented as a stored function.

The OpenODB inheritance model supports EXPRESS single and multiple inheritance. EXPRESS AND/OR inheritance can be represented by adjusting the types of instances using the OSQL “ADD TYPE” and “REMOVE TYPE” statements.

The OpenODB data model supports many features of EXPRESS, but not all of them. Some EXPRESS constructs must be simulated. OpenODB supports almost all of the EXPRESS primitive types. Logicals and enumerations must be simulated using object types. Instances of these object types were used to represent each of the enumerated values. Defined types of primitives are treated as the base primitive type. Corresponding EXPRESS and OpenODB types are shown in Table 4.2.

EXPRESS Types	OpenODB Types
REAL	float/double
INTEGER	integer
BOOLEAN	boolean
LOGICAL	class (three instances)
STRING(N)	char (var n)
BINARY(N)	binary (n div 8)
ENUMERATION	class (fixed instances)

Table 4.2 — Mapping from EXPRESS to the OpenODB Primitive Types

As with Oracle, EXPRESS selects are modeled as a tuple of possible types. Only the column for the type in use contains a value. All others are null.

OpenODB supports EXPRESS unordered aggregates (Bag and Set). Ordered aggregates (List and Array), are simulated as a bag of tuples with index and element columns. Nested aggregates are simulated as a bag of tuples with an element column and multiple index columns.

The OSQL language can define both computed and stored functions. EXPRESS inverse attributes map to functions containing OSQL queries. Mappings for EXPRESS derived attributes, local rules, and uniqueness constraints were not defined, but may be addressed by future projects.

4.3.3 Upload/Download Analysis

The upload/download implementations were straightforward to build and maintain. The upload and download programs were developed by writing programs for a group of sample types. Next, the programs were parameterized, and an EXPRESS compiler was modified to generate copies of the programs with specific types. The parameterized templates can be modified as needed for customized programs.

To understand the relative implementation costs, consider the amount of code required to build each system. The Oracle code generator required about 3000 lines of compiler extensions with another 2000 lines of template code. The OpenODB generator required 2000 lines of compiler extensions with 4000 lines of template code. As we will see, this is quite reasonable when compared to a direct SDAI implementation.

Code generation was used to take advantage of the Oracle query optimizer. A data-dictionary approach would require dynamic SQL, which is not as efficient and is not portable across relational systems. This was not an issue with the OpenODB implementation since all access is handled through dynamic SQL/OSQL, but code generation was still used because of its simplicity.

System	Software Written for Binding	Implementation Effort
OpenODB	OpenODB Oaci upload and download program templates, OSQL data definition templates, EXPRESS compiler generator extensions.	6000 lines
Oracle	Oracle Pro/C upload and download program templates, SQL data definition templates, EXPRESS compiler generator extensions.	5000 lines

Table 4.3 — Upload/Download Implementation Studies

The Oracle relational model and EXPRESS are significantly different, which forces the implementation software to do a large amount of assembly and disassembly when transferring data into and out of Oracle.

The OpenODB model is closer to EXPRESS, and the effort required to map between them is correspondingly lower. It should be noted, however, that the underlying OpenODB storage engine is based on relational technology. Therefore, the OpenODB object manager must perform the same assembly and disassembly as the Oracle implementation.

4.4 Cached SDAI Implementations

Next, we look at cached SDAI implementations based on Oracle and Versant. These bindings move data to and from an SDAI working-form cache. The Versant implementation may be used with any EXPRESS information model. The Oracle implementation only supports AP-203, although it was derived from the general-purpose upload/download system.

4.4.1 Oracle Cached SDAI

The cached SDAI interface to Oracle was constructed by modifying the upload/download system described in Section 4.3.1. The generated load and extract applications were modified to perform additional processing upon the data. This interface was constructed to run the AP-203 benchmarks described in the following chapters.

We already had an implementation of the benchmarks using the ROSE C++ library. The upload and download programs also used the library, so we merged the generated code into the benchmarks. The resulting benchmarks used data brought into memory from Oracle instead of a Part 21 file. The benchmarks only required read access, so the upload program was not merged, although this would not be a difficult task.

The implementation uses the same EXPRESS mapping described in Section 4.3.1. Since this implementation was based on the upload/download system, it can be classified as a code generation system. The generated code was hand-modified and currently supports only AP-203. Support of other AP's would not be difficult, but would require extending the generator software.

4.4.2 Versant Cached SDAI

The cached SDAI interface to Versant was constructed using a data-dictionary rather than code generation. All services are provided by a single library that a developer can link into an existing SDAI application.

The Versant interface extends an existing working-form SDAI C library [STI92a]. The original library uses a memory working-form and can read and write Part 21 exchange files, as well as create, delete, and manipulate data in main memory using SDAI operations. The interface adds the ability to transfer the memory working-form to and from a Versant database.

When transferring from memory cache to Versant database, the interface library connects to the database and compares the EXPRESS data-dictionary against the types defined in the Versant data-dictionary. Versant types for any missing EXPRESS definitions are created using the mapping described below. Next, Versant data objects are created and populated using data-dictionary calls to the Versant C API.

When loading the memory cache, the interface library compares the EXPRESS and Versant data-dictionaries to ensure a match between all types. If the EXPRESS data-dictionary is missing, it can be regenerated (with some information loss) from the Versant dictionary. Once the dictionaries have been synchronized, main memory objects are created by the SDAI library. The attribute values for these objects are extracted using calls to the Versant C API.

EXPRESS Mapping to Versant C++

The Versant data model is object-oriented and based loosely around C and C++. EXPRESS entities and inheritance relations are mapped to Versant classes in a manner similar to that defined for the SDAI C++ Binding [ISO95b]. Versant primitive types are based on

C and C++, with typedefs for hardware portability. The EXPRESS primitives are handled according to Table 4.4.

EXPRESS Types	Versant Types
REAL	o_float/o_double
INTEGER	o_4b
BOOLEAN	o_bool (typedef char)
LOGICAL	o_bool (typedef char)
STRING	Vstr of o_1b
BINARY	Vstr of o_1b
ENUMERATION	Vstr of o_1b

Table 4.4 — Mapping from EXPRESS to the Versant Primitive Types

Unions are not part of the Versant object model, so EXPRESS Selects are simulated as classes. A select class contains attributes for each type in the select, plus an additional attribute to indicate the active field. Aggregates are mapped to classes containing a size attribute and a dynamic vector based on the element type.

4.4.3 Cached SDAI Analysis

The Versant interface was built using data-dictionary calls instead of code generation. This simplified end-user operation of the interface, but made implementation slightly more complex and added to maintenance effort. Furthermore, the end-user cannot customize the interface.

A data-dictionary approach was chosen to avoid C++ class problems. Versant uses a C++ class library that is not directly compatible with the SDAI C and ROSE C++ libraries. Merging them would require changes to the base classes of each library. Such changes are known to be disruptive [Snyd86]; this has come to be known as the *fragile base class* problem [Mikh97]. Using data-dictionary functions, we were able to access Versant data without merging the class hierarchies.

System	Software Written for Binding	Implementation Effort
Oracle	Modify the upload/download software. Requires Oracle Pro/C upload and download program templates, SQL data definition templates, EXPRESS compiler generator extensions.	~100 lines changes plus 5000 lines upload/download software
Versant	Versant batch transfer software, data-dictionary synchronization. Integrate with existing working-form binding.	3000 lines

Table 4.5 — Cached SDAI Implementation Studies

Looking at the relative implementation costs, we see that the Versant load and unload routines required only a moderate amount of work. Adding the database features to the existing working-form binding required about 3000 lines of code.

The cached Oracle binding required even less implementation effort. We were able to leverage the existing upload/download implementation with only a few changes. The generated AP-203 upload and download programs were large (about 65,000 lines), but only a few hundred extra lines were needed to merge these programs with the working-form benchmark code.

Finally, the similarity between the Versant data model and EXPRESS simplified construction of a data-dictionary system. Only minimal manipulation was needed when moving data into and out of the cache. As we have noted, the Oracle relational model is quite different from EXPRESS and requires more manipulation. An Oracle data-dictionary approach would require the software to store enough information to reproduce the encoding at execution time. With code generation, the encoding can be determined by the EXPRESS compiler at generation time and the software can be simplified.

4.5 Direct SDAI Implementations

Finally, we look at direct SDAI implementations based on Oracle and ObjectStore. These bindings provide SDAI access using direct calls into the underlying database. The ObjectStore binding may be used with any EXPRESS information model while the Oracle binding is a hand-built research prototype that only supports AP-203.

4.5.1 Oracle Direct SDAI

The Oracle direct binding for AP-203 was implemented using code generation. Perl scripts, rather than an EXPRESS compiler, generate functions with embedded SQL calls. The binding only implemented SDAI features necessary for the AP-203 benchmarks described in following chapters. The benchmarks required attribute access and entity extent functions for a subset of the AP-203 types. A complete binding would have required significantly more effort.

This implementation used the same EXPRESS to SQL mapping as the other Oracle bindings. This enabled us to populate the database using the upload and download programs from Section 4.3.1.

Applications connect to the Oracle database in the usual way. Once connected, they find all objects of a particular type using entity extent functions. Objects are identified by a unique foreign key value. Since inherited attributes are normalized into separated tables, each SDAI object identifier is a foreign key that sews together many tables into a complete object. Each generated SDAI attribute access function performs a simple select on one of these tables and foreign keys to find a single row.

4.5.2 ObjectStore Direct SDAI

The ObjectStore implementation was developed as a direct SDAI binding. The interface was implemented using code generation. An EXPRESS compiler generates a C++ class for each structure in an information model. The ObjectStore database system intercepts pointer references to make C++ data persistent. This technique requires minimal changes to an existing application. The ROSE library was modified to create objects in persistent ObjectStore memory rather than transient heap memory.

An application developer can use the modified library to perform ROSE C++ operations on an ObjectStore database. Some ObjectStore functions must still be used to set special access points or control ObjectStore transactions.

EXPRESS Mapping to ObjectStore C++

The ObjectStore data model is C++. ObjectStore persistence is a property of memory allocation so any class library can be made persistent. We used the ROSE C++ classes described in [STI92b] as a basis for the EXPRESS structures. Entities and inheritance trees are mapped into a C++ class hierarchy using the approach defined by the SDAI C++ [ISO95b].

The EXPRESS primitives are mapped into C++ primitive types according to Table 4.6. The members of C++ **enum** types must be unique across all types, but EXPRESS allows many enumerations to contain the same member. A naming convention is used to make the C++ **enum** members unique.

Select types are modeled as subtypes of a special RoseUnion class. These subtypes encapsulate a normal C union attribute and keep track of which union member is currently set.

EXPRESS Types	ObjectStore Types
`REAL`	`float/double`
`INTEGER`	`int`
`BOOLEAN`	`BOOLEAN (typedef char)`
`LOGICAL`	`LOGICAL (typedef char)`
`STRING`	`char *`
`BINARY`	`BINARY class`
`ENUMERATION`	`enum`

Table 4.6 — Mapping from EXPRESS to the ObjectStore Primitive Types

Aggregates are modeled as parametrized C++ classes. These classes are derived from a hierarchy that starts at RoseAggregate and continues on to separate classes for List, Set, Bag, and Array. Subclasses of these are defined for each possible element type.

4.5.3 Direct SDAI Analysis

The ObjectStore binding was built using code generation. Code generation was mandatory because ObjectStore databases can only be defined or accessed using C++ classes. ObjectStore has a limited data-dictionary, but data must still be defined as classes.

The nature of ObjectStore enabled us to produce an SDAI direct binding with an artificially low level of implementation effort. ObjectStore implements its own virtual memory system and intercepts page-faults to make ordinary C++ applications persistent. By altering the memory allocation portions of the working-form SDAI library we were able to leverage many man-years of effort and over 40,000 lines of existing code. It is reasonable to expect a direct binding on a different C++ OODBMS — such as Versant — to require at least the same amount of effort as a working-form binding.

Because it operates directly upon the database, this interface can take advantage of transaction and locking features provided by the underlying system. Furthermore, this interface can be customized by extending the generated C++ classes with new member functions.

System	Software Written for Binding	Implementation Effort
ObjectStore	Modify existing working-form library to use ObjectStore memory allocation calls.	200+ lines
Oracle	Oracle Pro/C attribute access and entity extent functions, generated by Perl scripts. Only for 51/366 of the AP-203 entities. Requires upload and download programs and SQL data definitions for AP-203.	6500 lines (partial) 5000 lines (upload/ download tools)
	Full binding requires above services for all AP-203 types. Also attribute update functions, session model, Part 21 parser and writer, SQL data definition templates, EXPRESS compiler generator extensions.	91,000 lines (estimate for full binding)

Table 4.7 — Direct SDAI Implementation Studies

The Oracle direct binding only implemented the subset of the SDAI operations required for the benchmarks. Even with these simplifications, construction required a great deal of effort. The benchmarks required definitions for 51 entities. The attribute access and entity extent functions for these required 6500 lines of code. To provide the same for all 366 definitions in AP-203 would have required approximately seven times as much effort (45,500 lines). Adding support for update functions could double this total (91,000 lines). Finally, a binding should also provide a session model, Part 21 file handling, and other required services.

The Part 21 load and extract programs were reused from Section 4.3.1, but were built around a working-form binding. Providing the same capabilities directly on top of Oracle, without use of a working-form binding, would require an EXPRESS data-dictionary, Part

21 parser, and other facilities. Constructing these services could require a significant fraction of the effort associated with a working-form binding.

4.6 Implementation Summary

In this chapter, we examined the SDAI database implementations shown in Table 4.8. These implementations covered all three access styles.

Oracle and OpenODB Early-Bound Upload/Download	Oracle Early-Bound Cached	ObjectStore and Oracle Early-Bound Direct
Late-Bound Upload/Download	Versant Late-Bound Cached	Late-Bound Direct

Table 4.8 — SDAI Architectures Covered by the Implementation Studies

Table 4.9 summarizes the implementation costs for the systems as well as the amount of existing general-purpose code that could be reused. The alternate bindings (upload/download and cached) required a small amount of effort (~5000 lines) and were able to take advantage of a large amount of existing code (working-form binding and EXPRESS compiler).

The ObjectStore direct binding required a very small amount of effort (~200 lines) and was also able to take advantage of an existing working-form binding and EXPRESS compiler. However, these results are mostly due to the unique virtual-memory model used by ObjectStore.

Most database systems use a traditional API, like the Oracle Pro/C API, so the Oracle system is a better example of the implementation effort required for a direct binding. The

Oracle direct binding was only partially implemented and required a large amount of effort (11,500 lines). It required 6500 lines for access and entity extent functions covering some entities, plus 5000 lines for the upload/download and SQL definition programs. Expanding the binding to cover update functions, all AP-203 entities, a session model, and other requirements would require several times as much code.

System	Binding Architecture	Implementation Effort	Code Reuse
ObjectStore	Direct	200+ lines	40,000 lines (working-form binding)
OpenODB	Upload/ Download	6000 lines	40,000 lines (binding)
Oracle	Upload/ Download	5000 lines	40,000 lines (binding)
	Cached	5000+ lines	40,000 lines (binding)
	Direct	11,500 lines (partial + upload/ download tools) 91,000 lines (est. full binding)	none
Versant	Cached	3000 lines	40,000 lines (binding)

Table 4.9 — Implementation Cost Summary

We note that the cost for implementing each access style rises with the number of features it provides. The upload/download and cached bindings are inexpensive to produce and can reuse an existing working-form binding. These bindings provide SDAI access, but are not useful for situations requiring concurrent update. A direct binding is more costly, and generally cannot reuse code, but can make greater use of database features.

5 Operational Cost Benchmarks

5.1 Overview

To gain insight into operational costs, we must test SDAI bindings against a selection of real-world operations. The benchmarks described in this chapter are based on AP-203, which was the first of the STEP application protocols and contains information, such as CAD geometry and product configuration, that is common to all STEP models.

Section 5.2 describes AP-203 and identifies three categories of engineering information within the model. Section 5.3 through Section 5.5 discuss these aspects of AP-203 data and propose benchmark operations. These STEPStone benchmarks operate on information that is modeled in an existence-dependent style (PartStone, part versions), a navigational style (NURBStone, shape/geometry), and a mix of the two (BOMStone, bills of material).

5.2 The AP-203 Information Model

We looked briefly at AP-203 in Section 2.3.2. This was the first application protocol to be published as an ISO standard [ISO94f, PDES97] and has been used as the basis for many file exchange implementations. The scope of AP-203 is configuration-controlled 3-D product design data for mechanical parts and assemblies. This information includes the shape of parts, revision history, change process & documentation, part classification, approval, supplier & contract information, and security classification. The Units of Functionality (UOFs) defined by AP-203 are detailed in Table 5.1.

Unit of Functionality	Contents
shape (Total of six UOFs)	Geometry and topology of the part. UOFs cover advanced boundary representations, facetted b-reps, manifold surface with topology, non-topological surface & wireframe, and wireframe with topology
authorization	Part data approvals.
bill_of_material	Parts list for an assembly.
design_activity_control	Documents revision history of parts.
design_information	Material, surface, and process specifications.
effectivity	Usage of components in a product.
end_item_identification	Describes consumable goods (products).
part_identification	Defines parts and part versions.
source_control	Supplied part and supplier information.

Table 5.1 — AP-203 Units of Functionality

Once AP-203 was selected as the information model, the next task was to identify a representative set of benchmarks. Looking at the fourteen UOFs in AP-203, we note three different styles of engineering information:

- Navigational — Information such as the STEP Shape/geometry UOF. The references from entity to entity are in the same direction as the expected path of access. For example, *A* points to *B*, which then points to *C*. Access from *A* to *C* is simply a matter of following a chain of references.

- Existence-dependent — Information such as the STEP Part Identification UOF. References are usually in the direction opposite the expected path of access. For example, *B* points to *A*. Access from *A* to *B* requires a query or back-pointer. This style of modeling shows that *B* is existentially-dependent on *A*. If *A* does not exist, then *B* can not exist.

- Mixed — Information that contain both styles, such as the STEP Bill of Materials UOF.

For each information modeling style, we select a representative AP-203 UOF. In the following sections we define benchmarks that traverse and examine data from each UOF. We adopted names for the benchmarks based on the underlying UOF and the "stone" convention established by the Whetstone CPU benchmark [Curn76], and its many successors (Dhrystone, Khornerstone, etc.):

- PartStone — Part Identification UOF, existence-dependent definitions.
- BOMStone — Bill Of Materials UOF, mixed definitions.
- NURBStone — Shape UOFs, navigational definitions.

The benchmarks exercise data access capabilities. Future versions of the benchmarks could be extended to include update features, but this is beyond our current scope. The following sections describe each benchmark in detail, including the structure and use of the information, the algorithms, and the algorithm complexity.

5.3 PartStone — Part Identification

The STEP Part Identification UOF describes the concepts of product and product version. Since the STEP standard was designed to represent product data, all of the APs use the part identification definitions.

In engineering organizations, this type of information is often held by product data management (PDM) systems. A common operation on this type of data is to find all of the versions for a part. We use this operation as the basis for the PartStone benchmark.

The PartStone benchmark must traverse an AP-203 data set and print all versions of a part. The benchmark must repeat this operation for each part in the database. The following sections examine the structure of STEP part identification data as well as the algorithms we use to implement the benchmark traversal operation.

5.3.1 Application Objects

The Part Identification UOF defines three application objects:

```
PART
PART_VERSION
DESIGN_DISCIPLINE_PRODUCT_DEFINITION
```

The PART object describes an engineering artifact. A PART_VERSION describes a particular version of that artifact. A DESIGN_DISCIPLINE_PRODUCT_DEFINITION describes a context for the descriptions of aspects of a part. For example, AP-203 is used to describe the mechanical design characteristics of a part. Other APs might describe different characteristics. The PartStone benchmark only uses the PART and PART_VERSION application objects.

5.3.2 EXPRESS Definitions

The PART and PART_VERSION application objects are mapped into entities from the STEP integrated EXPRESS models as described in Table 5.2. An EXPRESS-G representation of these definitions is shown in Figure 5.1.

In the integrated EXPRESS schemas, PART application objects are represented as instances of *product*, while PART_VERSIONs are represented as *product definition forma-*

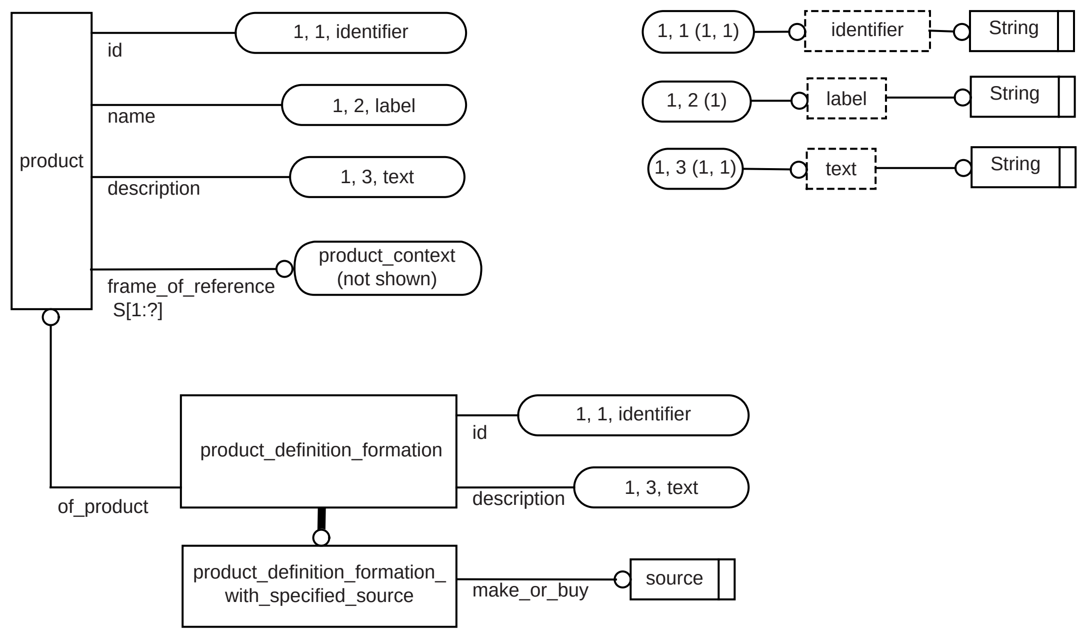

Figure 5.1 — EXPRESS-G Diagram of the Part Identification Entities

Application Object	**EXPRESS AP-203 Entity**
PART	product
PART_VERSION	product_definition_formation_with specified_source (abbreviated PDFWSS)

Table 5.2 — EXPRESS Entities for Part Identification

tion with specified source (abbreviated *pdfwss*). A pdfwss is tied back to a product via the *formation of* attribute. Figure 5.2 shows a small collection of parts and part versions, as well as the direction of access used by the PartStone benchmark.

The code fragment below shows how the Figure 5.2 data would be encoded within an AP-203 Part 21 file.

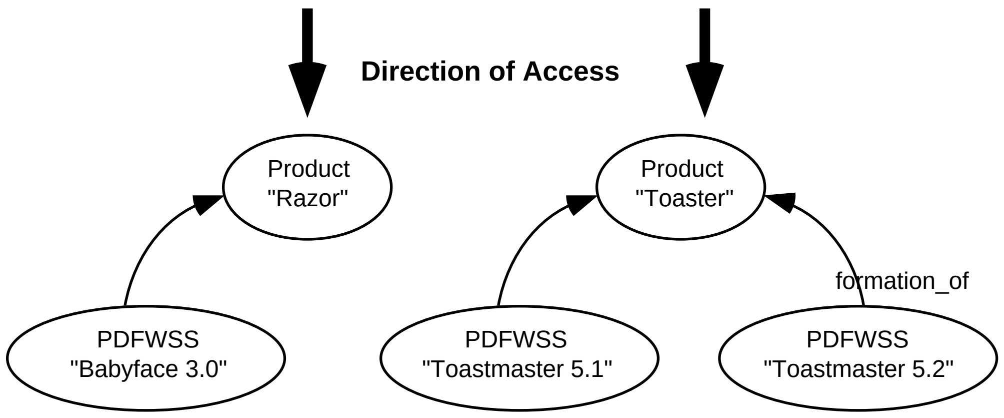

Figure 5.2 — Instance Diagram for Parts and Versions

```
#10=PRODUCT('PN-100','Razor','',$);
#11=PRODUCT_DEFINITION_FORMATION_WITH_SPECIFIED_SOURCE(
    'PN-100-1','BabyFace 3.0',#10,$);
#12=PRODUCT('PN-200','Toaster','',$);
#13=PRODUCT_DEFINITION_FORMATION_WITH_SPECIFIED_SOURCE(
    'PN-200-1','Toastmaster 5.1',#12,$);
#14=PRODUCT_DEFINITION_FORMATION_WITH_SPECIFIED_SOURCE(
    'PN-200-2','Toastmaster 5.2',#12,$);
```

5.3.3 Benchmark Operations

In the following pseudocode description of the PartStone benchmark, the PrintPart() and PrintVersion() functions print identifying information, such as a name or description attribute. In the following sections, all attribute access is indicated using a function of the form Get<att>().

The central benchmark operation is to print all versions for a single part. A pseudocode description of this operation is shown below:

```
void FindVersions (dataset : S, product : p)
begin
    foreach pdfwss : pdf in S do
        if  p = GetOfProduct(pdf) then PrintVersion(pdf)
end
```

The complete PartStone benchmark applies the FindVersions() algorithm to all of the products in a data set:

```
void PartStone (dataset : S)
begin
    foreach product : p in S do
    begin
        PrintPart(p)
        FindVersions (S, p)
    end
end
```

5.3.4 Complexity Analysis

Consider the PartStone algorithm. We determine the complexity of the algorithm given the following values:

```
P = number of products.
V = number of product versions.
```

For the purpose of analysis, it can be assumed that print functions and attribute access operate in constant time.

First, consider the FindVersions() function. This must examine each version in the database, so the function is O(V). Next, we call the function for each product in the database, which raises the complexity of the PartStone algorithm to O(PV). We observe that in practice, the number of versions for a particular part is normally small, and for the purpose of this argument, can be considered constant. So:

```
V = KP where K is the median number of versions
PartStone is O(PV) --> O(P * KP) --> O(P^2)
```

So the PartStone benchmark is O(P^2). It is tempting to improve the FindVersion() algorithm by constructing a sorted list of versions, or by some other global optimization. However, we must resist this temptation by recognizing that the benchmark must operate as if it were only printing the versions of a single part.

5.4 BOMStone — Bill of Materials

The STEP Bill of Materials UOF describes the structured list of materials or components required to build a product. A bill of material details all of the pieces that go into a product. It may call out quantities, sub-assemblies, variations within assemblies, allowable substitutions, and so forth. For example, the ingredients list from a recipe is a very simple bill of materials. A more complex one might show sub-assemblies, such as one often finds in the instructions for home-assembly furniture.

In engineering organizations, this type of information is maintained by product data management (PDM) systems and manufacturing requirements planning (MRP) systems. As one might imagine by the name, a common operation on this type of data is to print the list of assemblies and components —þthe bill of required materials for a product. This operation forms the basis of the BOMStone benchmark.

The BOMStone benchmark must traverse an AP-203 data set and print each assembly and its components. The following sections examine the structure of STEP bill of material data as well as the algorithms we use to implement the benchmark traversal operation.

5.4.1 Application Objects

The Bill of Materials UOF defines the following seven application objects. The list is indented to show when application objects are subtypes derived from a more general type of application object:

```
ENGINEERING_ASSEMBLY
    ENGINEERING_NEXT_HIGHER_ASSEMBLY
    ENGINEERING_PROMISSORY_USAGE

ALTERNATE_PART
COMPONENT_ASSEMBLY_POSITION
ENGINEERING_MAKE_FROM
SUBSTITUTE_PART
```

ENGINEERING_ASSEMBLY describes a parent/child relationship between an assembly and one of its components. ENGINEERING_NEXT_HIGHER_ASSEMBLY is a more specific type of relation which allows us to specify a name for the component, quantity, and unit of measure. ENGINEERING_PROMISSORY_USAGE describes a relationship between components and a sub-assembly that has not yet been defined.

ENGINEERING_MAKE_FROM, ALTERNATE_PART, and SUBSTITUTE_PART convey other relationships between the parts. These indicate how some parts serve as raw materials for others, or how they may act as replacements under some circumstances.

Finally, the COMPONENT_ASSEMBLY_POSITION relationship associates a geometric transform with a part to specify its physical location within an assembly.

The BOMStone benchmark uses ENGINEERING_ASSEMBLY application objects. The other application objects convey important information about individual components within an assembly, but do not contribute to the description of the assembly structure.

5.4.2 EXPRESS Definitions

The ENGINEERING_ASSEMBLY application objects are mapped into entities from the STEP integrated EXPRESS models as described in Table 5.2. Figure 5.3 shows the definitions as an EXPRESS-G diagram.

Application Object	EXPRESS AP-203 Entity
ENGINEERING_ASSEMBLY	assembly_component_usage (supertype)
ENGINEERING_NEXT HIGHER_ASSEMBLY	next_assembly_usage_occurrence
ENGINEERING PROMISSORY_USAGE	promissory_usage_occurrence

Table 5.3 — EXPRESS Entities for Bill of Material Assembly Structures

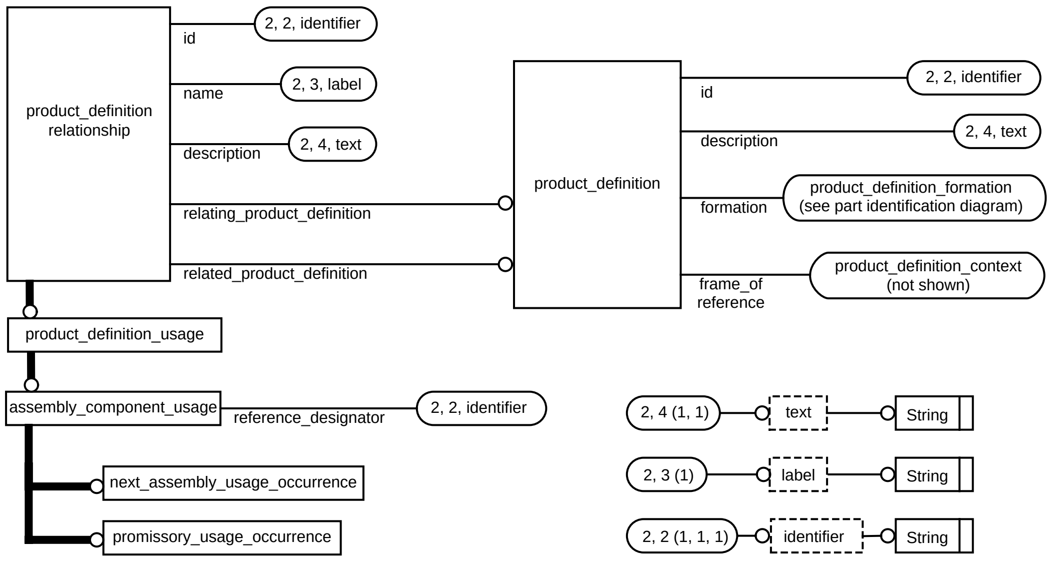

Figure 5.3 — EXPRESS-G Diagram of the Bill of Material Engineering Assembly Entities

These relationships connect *product definition* objects. A product definition object describes one aspect of a product version, such as the shape. Using multiple product definitions for each product, we could represent different types of assemblies. They could show electrical connectivity, physical arrangement, operational units, or manufacturing sub-assemblies. AP-203 describes mechanical assemblies, but an electrical AP could use a similar mechanism to describe functional components. The product definitions are tied together using the following two attributes:

- *relating product definition* — Points “upwards” in the assembly to the enclosing product definition. If we were relating the wheels of a car to an entire car, this would point at the car definition.

- *related product definition* — Points “downwards” in the assembly to the component product definition. In the car example, this would point to the wheel definition.

Each assembly component usage allows a reference designator, which is used to distinguish between multiple uses of the same component. In our car example, we have four assembly component usages, each relating the same wheel definition to the car definition. The reference designator indicates which is the right front wheel, left front, and so on.

These relationships connect the components of an assembly to the whole. Used recursively, each component can act as a sub-assembly related to various sub-components. Figure 5.4 shows a two level assembly, where an automobile is built from four copies of a wheel sub-assembly. The wheel sub-assembly is built from a rim, a tire, and a hubcap. The diagram also shows the direction of access used by the BOMStone benchmark.

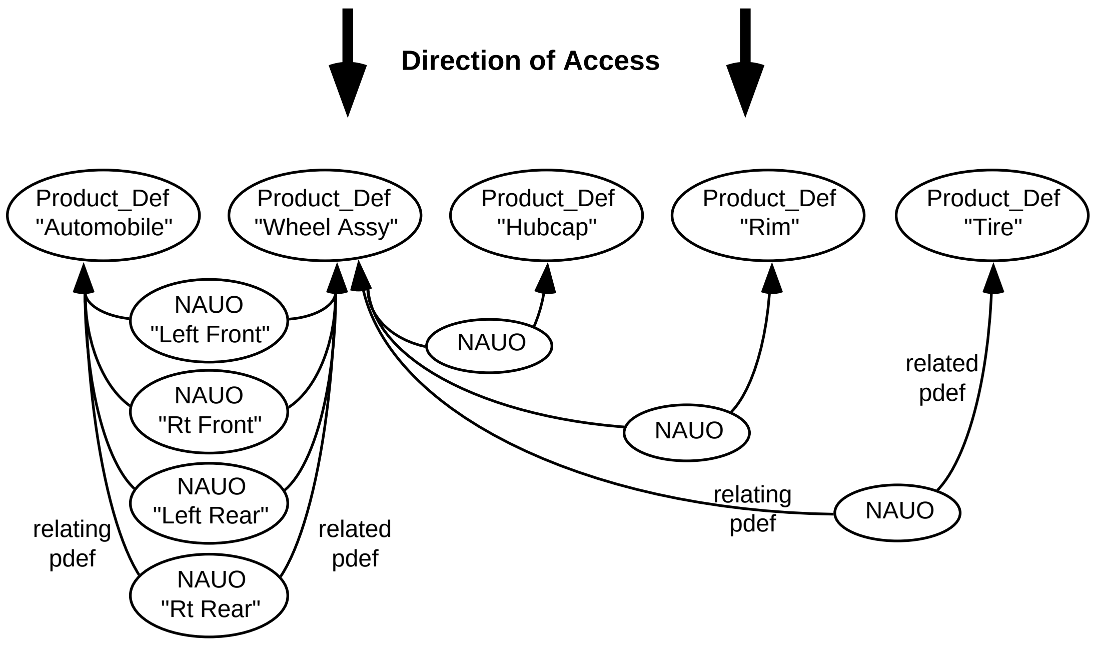

Figure 5.4 — Instance Diagram of a Two Level Automobile Assembly

The code fragment below shows how the Figure 5.4 data would be encoded within an AP-203 Part 21 file.

```
#10=PRODUCT('PN-001','Automobile','',$);
#11=PRODUCT_DEFINITION_FORMATION_WITH_SPECIFIED_SOURCE(
   'PN-001-1','Automobile v1',#10,$);
#12=PRODUCT_DEFINITION('initial','Automobile definition',#11,$);
#13=NEXT_ASSEMBLY_USAGE_OCCURRENCE('','','',#12,#22,'Left Front');
#14=NEXT_ASSEMBLY_USAGE_OCCURRENCE('','','',#12,#22,'Rt Front');
#15=NEXT_ASSEMBLY_USAGE_OCCURRENCE('','','',#12,#22,'Left Rear');
#16=NEXT_ASSEMBLY_USAGE_OCCURRENCE('','','',#12,#22,'Rt Rear');

#20=PRODUCT('PN-002','Wheel Assy','',$);
#21=PRODUCT_DEFINITION_FORMATION_WITH_SPECIFIED_SOURCE(
   'PN-002-1','Wheel Assy v1',#20,$);
#22=PRODUCT_DEFINITION('initial','Wheel Assy definition',#21,$);
#23=NEXT_ASSEMBLY_USAGE_OCCURRENCE('','','',#22,#32,'');
#24=NEXT_ASSEMBLY_USAGE_OCCURRENCE('','','',#22,#42,'');
#25=NEXT_ASSEMBLY_USAGE_OCCURRENCE('','','',#22,#52,'');

#30=PRODUCT('PN-003','Hubcap','',$);
#31=PRODUCT_DEFINITION_FORMATION_WITH_SPECIFIED_SOURCE(
   'PN-003-1','Hubcap v1',#30,$);
#32=PRODUCT_DEFINITION('initial','Hubcap definition',#31,$);

#40=PRODUCT('PN-004','Rim','',$);
#41=PRODUCT_DEFINITION_FORMATION_WITH_SPECIFIED_SOURCE(
   'PN-004-1','Rim v1',#40,$);
#42=PRODUCT_DEFINITION('initial','Rim definition',#41,$);

#50=PRODUCT('PN-005','Tire','',$);
#51=PRODUCT_DEFINITION_FORMATION_WITH_SPECIFIED_SOURCE(
   'PN-005-1','Tire v1',#50,$);
#52=PRODUCT_DEFINITION('initial','Tire definition',#51,$);
```

5.4.3 Benchmark Operations

The BOMStone benchmark must traverse a data set and print each assembly and its components as an indented list. As we saw in the previous section, the assembly structure is stored as *assembly component usage* instances (actually as the *next assembly usage occurence* subtype). These objects are the edges in the assembly graph.

The central benchmark operation is to find all components of a single product. This requires searching through all assembly component usages for those linked to the product via the *relating product definition* attribute. From this set, we can find the components by following the *related product definition* attribute. We continue recursively to find the subcomponents of each component. For each product, we print some identifying information and indent according to the product's position in the assembly tree. The pseudocode description of the algorithm is shown below:

```
void FindAssembly (product_definition  pdef, integer depth)
begin
    PrintIndent (depth)     /* indent according to depth */
    PrintProductDef (pdef)  /* print product and version name */

    /* FIND AND RECURSIVELY PRINT ALL SUB-COMPONENTS */
    foreach assembly_component_usage : acu in S do
    begin
        if pdef = GetRelatingProductDefinition(acu)
        then FindAssembly (GetRelatedProductDefinition (acu), depth+1)
    end
end
```

To print an entire assembly, we must determine which product represent the complete assembly — the "root" of the assembly tree. To find all top-level products, we search for product definitions that are not a sub-component of another product. In terms of the EXPRESS definitions, a product is at the top level if there exist no assembly component usages linking the product as component to a larger assembly. A pseudocode description of the algorithm is shown below.

```
List FindTopLevel (dataset : S)
begin
    List : toplevel
    foreach product_definition : pdef in S do
    begin
        foreach assembly_component_usage : acu in S do
        begin
            /* does it belong to an another assembly? */
            if pdef = GetRelatedProductDefinition (acu)
            then next product_definition
        end
```

```
            /* not a component of anything */
            Append (toplevel, pdef)
        end
        return toplevel
    end
```

The BOMStone algorithm combines these functions to print all of the assemblies in a data set. The complete BOMStone algorithm is shown below:

```
void BOMStone (dataset : S)
begin
    foreach product_definition : pdef in FindTopLevel(S) do
        FindAssembly (pdef, 0);
end
```

5.4.4 Complexity Analysis

The BOMStone algorithm operates on assembly structures. We can view an assembly structure as a directed graph, with product definitions as the nodes of the graph and assembly component usages as the edges:

```
P = number of products (nodes)
A = number of assembly component usages (edges)
```

In the general case, the graph is acyclic since a mechanical part cannot physically contain itself as a component. Is could also be a multigraph, if components participate in more than one assembly (as in Figure 5.4). For the purpose of our analysis we consider the simplified case where assembly is a tree. Given these assumptions, we can represent A in terms of P:

```
A = P - 1     for a one connected component (top-level assembly)
A = P - T     for T connected components
```

Consider the FindTopLevel() function. For each product definition, it examines all assembly component usages, so the function will require O(PA), or O(P^2), operations.

The FindAssembly() function examines all assembly component usages so it is O(A), or just O(P). Given that the input data set consists of one or more trees, the FindAssembly() function will be called once for each product definition. Therefore over the entire program run it will require $O(P^2)$ operations. The entire BOMStone algorithm is $O(P^2)$.

5.5 NURBStone — Part Geometry

The STEP Shape UOFs describe the geometric and topological aspects of a product. Since geometry and topology are basic physical properties, these definitions are used by almost all of the STEP APs. There are several Shape UOFs, each of which supports different mathematical representations of geometry, such as planar facets or NURB surfaces.

In engineering organizations, this type of information is usually created and managed by CAD systems. When a CAD system reads a shape description, it must traverse the entire geometric model in order to create a visual representation of the model. We use this traversal as the basis for the NURBStone benchmark.

The NURBStone benchmark must traverse an AP-203 data set and print the components of each geometric definition in a recursive-descent manner. The following sections examine the structure of STEP shape data as well as the algorithms used to implement the benchmark traversal operation.

5.5.1 Application Objects

The six Shape UOFs define the following application objects:

```
SHAPE
SHAPE_ASPECT
GEOMETRIC_MODEL_REPRESENTATION
    ADVANCED_BOUNDARY_REPRESENTATION
    FACETTED_B_REP
```

```
MANIFOLD_SURFACE_WITH_TOPOLOGY
NON_TOPOLOGICAL_SURFACE_AND_WIREFRAME
WIREFRAME_WITH_TOPOLOGY
```

The Shape UOF defines the SHAPE, SHAPE_ASPECT, and GEOMETRIC_MODEL REPRESENTATION objects, while the other UOFs constrain the type of geometric definitions permitted for describing a shape. The NURBStone benchmark only makes use of the ADVANCED_BOUNDARY_REPRESENTATION definition.

The SHAPE application object represents the physical form of a part, which is mathematically defined by one or more GEOMETRIC_MODEL_REPRESENTATION objects. A SHAPE_ASPECT calls out a portion of a shape to indicate subdivisions or attach additional specifications. A SHAPE_ASPECT also has one or more associated GEOMETRIC MODEL_REPRESENTATION objects.

5.5.2 EXPRESS Definitions

The ADVANCED_BOUNDARY_REPRESENTATION application object is mapped into entities from the STEP integrated EXPRESS models as described in Table 5.2. An EXPRESS-G representation of these definitions is shown in Figure 5.5.

Application Object	EXPRESS AP-203 Entity
GEOMETRIC_MODEL REPRESENTATION	shape_representation (supertype)
ADVANCED_BOUNDARY REPRESENTATION	advanced_brep_shape_representation

Table 5.4 — EXPRESS Entities for Bill of Material Assembly Structures

The *shape representation* entity contains a list of geometric items, as well as a name and context. The subtypes of this, such as *advanced brep shape representation*, do not add any attributes. They merely constrain the contents of the geometric items list to certain types

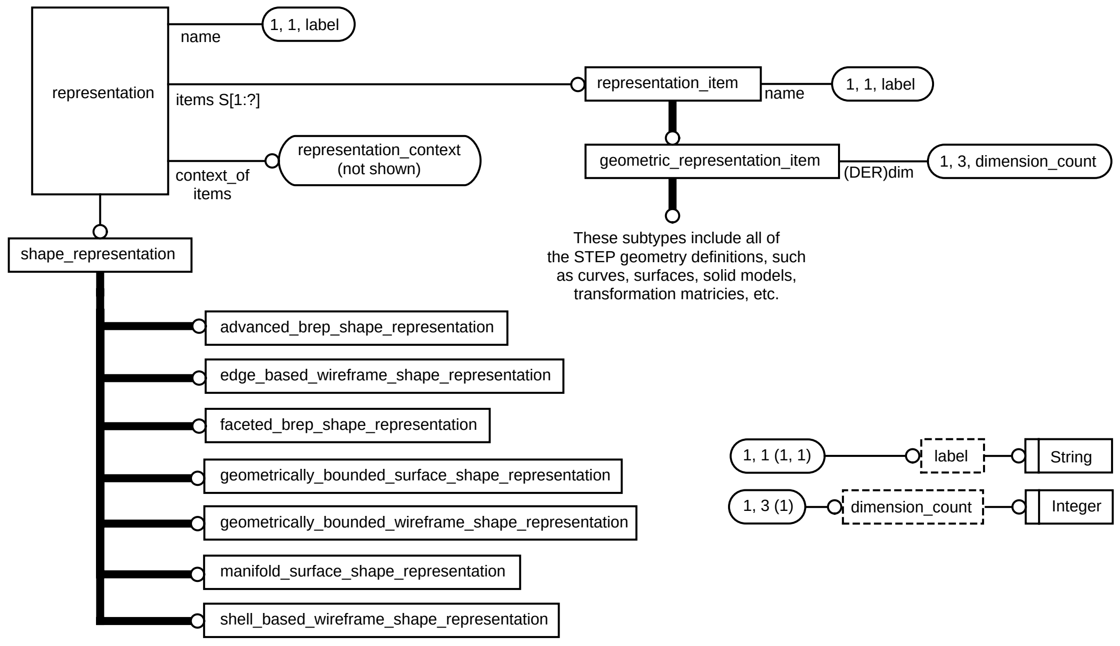
representation
name
1, 1, label
items S[1:?]
representation_item
name
1, 1, label
context_of items
representation_context (not shown)
geometric_representation_item
(DER)dim
1, 3, dimension_count
These subtypes include all of the STEP geometry definitions, such as curves, surfaces, solid models, transformation matricies, etc.
shape_representation
advanced_brep_shape_representation
edge_based_wireframe_shape_representation
faceted_brep_shape_representation
geometrically_bounded_surface_shape_representation
geometrically_bounded_wireframe_shape_representation
manifold_surface_shape_representation
shell_based_wireframe_shape_representation
1, 1 (1, 1)
label
String
1, 3 (1)
dimension_count
Integer

Figure 5.5 — EXPRESS-G Diagram of the Geometric Model Representation Entities

of geometry. Figure 5.6 shows the components of a manifold solid B-REP shape description, as well as the direction of access used by the NURBStone benchmark.

The code fragment below shows the shape definition for a toroid, containing all of the elements from Figure 5.6. The data is an advanced B-REP solid, which means that it contains geometric surface details (*toroidal surface*) and topological details (*vertex loop*). To be complete, the description must specify units of measure and tolerance (instances #27-#37), as well as a location within the global coordinate system (the *axis2 placement 3d*).

```
#10=ADVANCED_BREP_SHAPE_REPRESENTATION('',(#11,#23),#27);
#11=MANIFOLD_SOLID_BREP('',#12);
#12=CLOSED_SHELL('',(#13));
#13=ADVANCED_FACE('',(#14),#18,.T.);
#14=FACE_BOUND('',#15,.T.);
#15=VERTEX_LOOP('',#16);
#16=VERTEX_POINT('',#17);
```

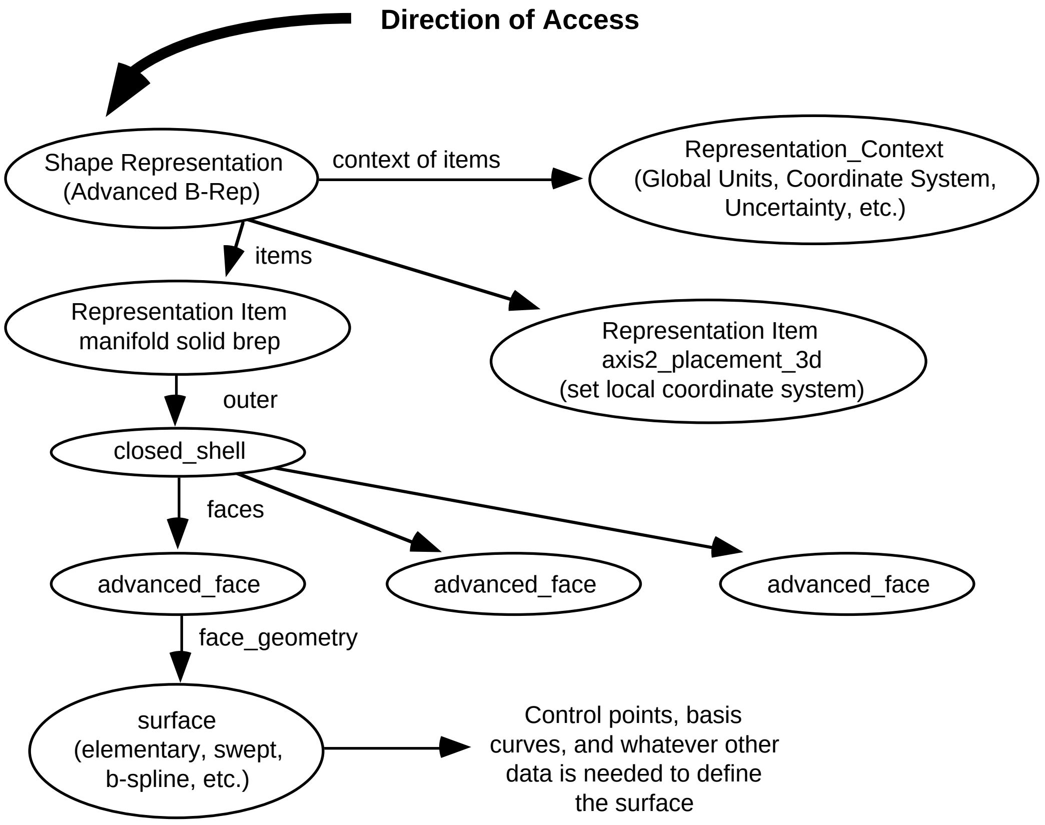
Direction of Access
Shape Representation (Advanced B-Rep)
context of items
Representation_Context (Global Units, Coordinate System, Uncertainty, etc.)
items
Representation Item manifold solid brep
Representation Item axis2_placement_3d (set local coordinate system)
outer
closed_shell
faces
advanced_face
advanced_face
advanced_face
face_geometry
surface (elementary, swept, b-spline, etc.)
Control points, basis curves, and whatever other data is needed to define the surface

Figure 5.6 — Instance Diagram for the Major Components of a Manifold Solid B-REP Shape Description

```
#17=CARTESIAN_POINT('',(-22.5,-0.500000000000002,-37.5));
#18=TOROIDAL_SURFACE('',#19,10.,5.);
#19=AXIS2_PLACEMENT_3D('',#20,#21,#22);
#20=CARTESIAN_POINT('',(-7.5,-0.500000000000002,-37.5));
#21=DIRECTION('',(0.,0.,-1.));
#22=DIRECTION('',(-1.,0.,0.));
#23=AXIS2_PLACEMENT_3D('',#24,#25,#26);
#24=CARTESIAN_POINT('',(0.,0.,0.));
#25=DIRECTION('',(0.,0.,1.));
#26=DIRECTION('',(1.,0.,0.));

/* Sets up units (inches, degrees, steradians) and allowable
 * measurement tolerance (3.9e-7 inches) for the geometry
 */
#27=(
    GEOMETRIC_REPRESENTATION_CONTEXT(3)
    GLOBAL_UNCERTAINTY_ASSIGNED_CONTEXT((#28))
    GLOBAL_UNIT_ASSIGNED_CONTEXT((#29,#33,#37))
    REPRESENTATION_CONTEXT('ph1m4-ug','COMPONENT_PART')
);
#28=UNCERTAINTY_MEASURE_WITH_UNIT(LENGTH_MEASURE(3.93700787401575E-07),#29,
   'MODEL_ACCURACY','Maximum Tolerance applied to model');
#29=( CONVERSION_BASED_UNIT('INCH',#31) LENGTH_UNIT() NAMED_UNIT(#30) );
#30=DIMENSIONAL_EXPONENTS(1.,0.,0.,0.,0.,0.,0.);
#31=LENGTH_MEASURE_WITH_UNIT(LENGTH_MEASURE(25.4),#32);
#32=( LENGTH_UNIT() NAMED_UNIT(*) SI_UNIT(.MILLI.,.METRE.) );
#33=( CONVERSION_BASED_UNIT('DEGREE',#35) NAMED_UNIT(#34) PLANE_ANGLE_UNIT() );
#34=DIMENSIONAL_EXPONENTS(0.,0.,0.,0.,0.,0.,0.);
#35=PLANE_ANGLE_MEASURE_WITH_UNIT(PLANE_ANGLE_MEASURE(0.0174532925),#36);
#36=( NAMED_UNIT(*) PLANE_ANGLE_UNIT() SI_UNIT($,.RADIAN.) );
#37=( NAMED_UNIT(*) SI_UNIT($,.STERADIAN.) SOLID_ANGLE_UNIT() );
```

5.5.3 Benchmark Operations

The NURBStone benchmark must traverse and print each shape description in a data set. As shown in Figure 5.6, a STEP shape description forms a tree of instances, with an instance of *shape representation* at the root. As one proceeds from the root, complex geometric types (*manifold solid brep*) refer to component instances (*closed shell*, *advanced face*) which refer to surfaces, and so on. The leaves are geometric primitives that refer to nothing else (*cartesian point*).

The NURBStone benchmark performs a depth-first search of the shape instance tree. Each node could be a different type of instance, so the algorithm may require functions for each possible entity type. The final benchmark resembles recursive-descent parsing algorithm.

Some shape representations contain topology data, which connects surfaces to form solid models. The NURBStone benchmark has been defined to traverse only surface geometry instances. However, future work could extend the scope of the benchmark to also traverse the topology instances.

Using the convention established in the Section 5.3.3 and Section 5.4.3, we assume a Print<type>() function for each entity type. We also define a Traverse<type>() function for each non-leaf entity type. The general outline for a traverse function is shown below:

```
void Traverse<Type> (<Type> : obj)
begin
    Print<Type>(obj)
    foreach instance : ref referred to by obj do
        Traverse<RefType> (ref)
end
```

The function prints identifying information about the instance, such as a name, coordinates, or a radius. Next, the function examines all attributes and calls the traversal function (or print function the case of leaf types) for each instance referred to by an attribute.

The following example shows the traversal function for *shape representation*. This entity has a name attribute as well as references to a *representation context* and a list of *representation items*. The EXPRESS-G for *shape representation* can be found in Figure 5.5.

```
void TraverseShapeRep (shape_representation : rep)
begin
    PrintShapeRep (rep)   /* prints name */
    TraverseRepContext (GetContextOfItems(rep))
    foreach representation_item : repitem in GetItems(rep) do
        TraverseRepItem (repitem)
end
```

The NURBStone benchmark defines traversal and print function for the AP-203 entities shown in Table 5.5.

axis1_placement	axis2_placement
axis2_placement_2d	axis2_placement_3d
b_spline_surface	b_spline_surface_with_knots
cartesian_point **	circle
conic	closed_shell
conical_surface	context_dependent_unit**
conversion_based_unit**	curve
curve_bounded_surface	curve_replica
cylindrical_surface	direction **
elementary_surface	ellipse
face_surface	geometric_representation_context
global_uncertainty_assigned_context	global_unit_assigned_context
hyperbola	line
manifold_solid_brep	measure_value **
named_unit **	offset_curve_2d
offset_curve_3d	parabola

Table 5.5 — Entity Types Traversed by the NURBStone Benchmark

pcurve	plane
rational_b_spline_surface	rectangular_composite_surface
rectangular_trimmed_surface	representation_context
representation_item	shape_representation
si_unit **	spherical_surface
surface	surface_curve
surface_of_linear_extrusion	surface_of_revolution
swept_surface	toroidal_surface
unit **	vector

Leaf entity types are denoted by **

Table 5.5 — Entity Types Traversed by the NURBStone Benchmark

Some of the entities in Table 5.5 are supertypes of other entities. Traversal functions for supertypes must exhibit polymorphism, i.e. they must examine the instance type and call a more specific traversal function if required. The following example shows the traversal function for *surface*, which is the supertype of all geometric surface types. The pseudocode describes the algorithm using if-then clauses. An implementation might provide polymorphism in a more efficient manner, such as C++ virtual functions.

```
void TraverseSurface (surface : s)
begin
    if  IsElementarySurface(s)  then
        TraverseElementarySurface(s)
    else if IsSweptSurface(s) then
        TraverseSweptSurface(s)
    else if IsBSplineSurface(s) then
        TraverseBSplineSurface(s)
    else if IsCurveBoundedSurface(s) then
        TraverseCurveBoundedSurface(s)
    else if IsRectangularTrimmedSurface(s) then
        TraverseRectangularTrimmedSurface (s)
    else if IsRectangularCompositeSurface(s) then
        TraverseRectangularCompositeSurface (s)
end
```

The NURBStone algorithm combines all of these functions to print each shape representations in the data set. The depth-first traversal begins with *shape representation*. The top-level benchmark algorithm is shown below:

```
void NURBStone (dataset : S)
begin
```

```
        foreach shape_representation : rep in S do
            TraverseShapeRep (rep);
    end
```

5.5.4 Complexity Analysis

Maintaining our initial assumption that the shape representation instances are a tree, we note that the NURBStone depth-first traversal algorithm touches each instance once. In addition, the print actions for each object are performed in constant time (the exact time may change from type to type.)

So, although a shape representation contains many different types of instances, the benchmark complexity depends only on the total number of instances in the data set. The NURBStone complexity is O(N).

If instances were shared between shapes, the shape representation would not be a tree, but rather an acyclic directed graph. The benchmark algorithm would visit shared instances more than once, raising the complexity. We could restore linear complexity by “marking” each instance as it is visited, but we avoid this change since storing the mark would require updating the database. As stated in Section 5.2, the benchmark scope is limited to access only. Of course, we could keep “mark” information in a local data structure with favorable performance, such as a hash table. This would add to the resource usage of the algorithm, but in the case of a hash table, would not affect the complexity.

In practice, we observe that instances are rarely shared between shapes. Only “housekeeping” data, such as unit declarations and other representation context information, are ever shared. In this work, we enforce the constraint against shared objects and so are not required to store “mark” information.

6 Benchmark Results

6.1 Overview

To gain insight into operational costs, the STEPStone benchmark algorithms were run against SDAI implementations built on a relational database, object-oriented database, and a main memory cache. In addition, we looked at load/extract times and explored the effect of optimizations on each implementation. Timing measurements were made with data sets ranging in size from 100-100k objects. These measurements required about 500 program runs and 220+ hours of compute time on a SPARC 20.

The remainder of this section describes the test systems and timing methods. Section 6.2 through Section 6.4 describes the results from the PartStone, BOMStone, and NURBStone benchmarks. Section 6.5 describes the results of database load and extract tests.

6.1.1 SDAI Test Systems

The measurements in this chapter were made using STEPStone benchmarks built on three different SDAI implementations. The benchmarks were implemented on ObjectStore using the direct binding described in Section 4.5.2, on Oracle using the direct binding described in Section 4.5.1, and on a main-memory cache using the ROSE working-form binding described in Section 4.2.5.

All timings were done on a Sun SPARC 20 with 128 megabytes of memory. The machine was running the Solaris v2.4 operating system. The database versions were Oracle v7.3 and ObjectStore v4.0.2. The SDAI implementations were built using ST-Developer v1.6, ST-Oracle v1.6.0, and ST-ObjectStore v1.6.0.

6.1.2 Timing Methods

All times measured were total elapsed (wall clock) times rather than CPU usage times. Wall clock times were used to ensure a fair comparison among the benchmarks. Database systems often connect to server processes that perform much of the database work. Unfortunately, the resource usage system calls do not return the CPU usage of server processes. They only return the CPU usage of the initial process.

The total elapsed time is unaffected by distributed processing and provides an accurate measure of real world performance. The wall clock time will be larger than the actual CPU usage, but should remain proportional to the CPU usage as long as care is taken to perform all benchmarks under identical conditions. The measurements in this work were performed only during periods of light machine load. In addition, the database servers were restarted before each program run.

All measurements were rounded to the nearest second. When multiple measurements were present, they were averaged to the nearest tenth of a second.

6.1.3 Data Sets

The measurements explored the access behavior of the SDAI implementations for different types of data. It was necessary to run benchmarks with data sets of varying sizes, while maintaining the essential characteristics as the data set sizes increased.

Programs were developed to generate data sets of arbitrary size for each benchmark. These programs are discussed with the results of each benchmark. The programs produced separate files for each data set. The working-form benchmarks read the files, but the Oracle and ObjectStore benchmarks read databases built from the files. The Oracle databases were created using the upload/download tools described in Section 4.3.1. Before loading, all tables in the database were dropped, the schema was reloaded, and the Oracle cache daemon processes were restarted. Similarly, ObjectStore databases were removed and recreated before each benchmark run.

6.2 PartStone Results

The PartStone benchmark was implemented using each of the three SDAI systems. Timing measurements were made for each system using a series of data sets containing between 100 and 20,000 part objects. The composition of the data sets are described in Section 6.2.1. In addition to the basic algorithm, we tested optimizations based on the unique characteristics of each system. The optimizations are discussed in Section 6.2.3. The PartStone timings are shown Figure 6.1 and Figure 6.2.

6.2.1 Part Identification Test Data

The *mkpart* program was developed to generate data sets with an arbitrary number of parts and versions. To reduce the number of variables, the generated data sets contained a constant number of versions per part. As described in Section 5.3.4, this allows us to express the complexity of the PartStone algorithm in terms of the number of parts.

For the measurements in this work, the data sets contained three versions per part. Ten PartStone data sets were used. The sizes were: 100, 500, 1000, 1500, 2000, 2500, 5000, 10000, 15000, and 20000 part objects.

6.2.2 PartStone Timings

The basic PartStone algorithm was tested against the Oracle, ObjectStore and working-form implementations. The Oracle timings were performed with and without extra indices on heavily-used columns.

The ObjectStore and working-form implementations had similar performance. The times for both systems exhibit growth characteristic of an O(N^2) algorithm. The Oracle implementation was significantly slower, and we were unable to complete measurements on data sets with more than 500 parts. Addition of indices improved the Oracle behavior slightly, but not enough to complete all measurements. The extreme behavior of the Oracle implementation suggests a higher degree of complexity than O(N^2).

The Oracle indices were added on the keys of the *product*, *product definition formation*, and *product definition formation with specified source* tables, as well as the *of product* attribute of *product definition formation*.

6.2.3 PartStone Optimizations

Additional timings explored the effect of optimizations based on unique features of each system. The focus of most optimizations was the FindVersions() operation. This function traverses over all versions (*product definition formation* objects) to find ones that reference a particular product. In EXPRESS, this information can be found through a call to the Usedin() function. Originally, the SDAI specification did not provide a Usedin() operation, but recent versions have added optional support for it.

Each implementation of the PartStone benchmark was modified to perform the Usedin() operation as efficiently as possible for the *of product* relationship between *pdfwss* and *product*. The C++ classes used by the ObjectStore and working-form implementation were modified to hold back-pointers for the relationship. The Oracle implementation was modified to perform a single SQL join rather than traverse an entire entity extent using SDAI functions. In addition to Usedin() optimizations, multiple SDAI “get attribute” calls were collapsed into a single SQL query wherever possible.

Relational Optimizations

As mentioned above, the Oracle implementation was modified to perform the FindVersions() operation using a single SQL join. In addition, other SDAI calls were collapsed into single SQL queries where possible.

The modifications had a dramatic effect. The time required to process 500 parts went from eight hours down to ten seconds. Performance improved to roughly linear behavior, and the benchmark ran to completion in under an hour on even the largest data set. Adding extra indices to the tables did not improve performance. In fact, the benchmark was slightly slower with indices. To address concerns that this behavior may indicate a poor choice of indices, the timings were repeated with other index combinations. Times varied slightly, but were consistent with the initial observations.

ObjectStore and Working-Form Optimizations

The ObjectStore and working-form systems do not have general purpose query facilities. Instead, they must access data by type (find all objects of a particular type) or navigation (follow a pointer from one object to another). In place of a query facility, these systems use C++ classes that can be extended with additional functions or data.

To improve the FindVersions() function, the C++ class for *product* entities was modified to keep a list of back-pointers for the *of product* attribute of *pdfwss*. The back-pointers were initialized by traversing over all *pdfwss* objects at the beginning of the benchmark run. The modified algorithm is shown below:

```
void PartStone_backpointer (dataset : S)
begin
    foreach pdfwss : pdf in S do
```

```
        AppendOfProductBackpointers (GetOfProduct (pdf), pdf)

    foreach product : p in S do
    begin
        PrintPart(p)
        FindVersions_backpointer (S, p)
    end
end

void FindVersions_backpointer (dataset : S, product : p)
begin
    foreach pdfwss : pdf in GetOfProductBackpointers(p) do
        PrintVersion(pdf)
end
```

This optimization reduces the complexity of the PartStone benchmark to linear time. The initialization of the back-pointers requires O(V) time, but we can find the versions for a part in constant time. Summed over all parts, the FindVersions() function will touch each version once, requiring O(V) operations. Since we know that V can be represented as a constant time P, the optimized PartStone algorithm is O(P).

The modifications had a large effect on the observed benchmark times. As predicted, the times for the ObjectStore and working-form implementations grew in a linear fashion. The percentage difference between the two systems was much greater with the optimized benchmark than with the original. On the larger data sets, the difference between the original benchmarks was a few percent at best, while with the optimized benchmarks, the working-form implementation ran in half of the time of the ObjectStore implementation.

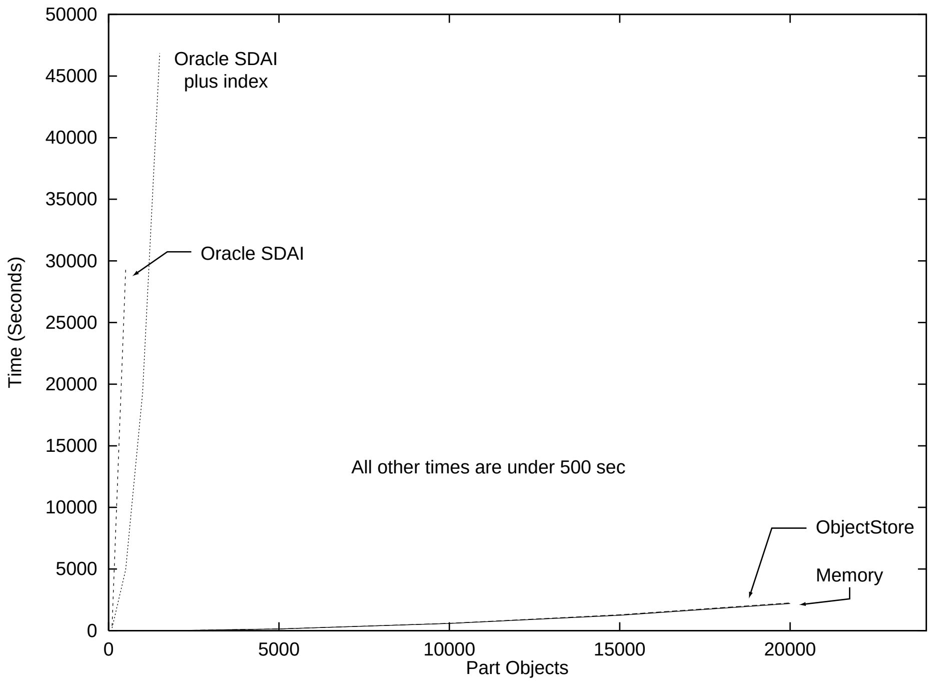

Figure 6.1 — PartStone Timings

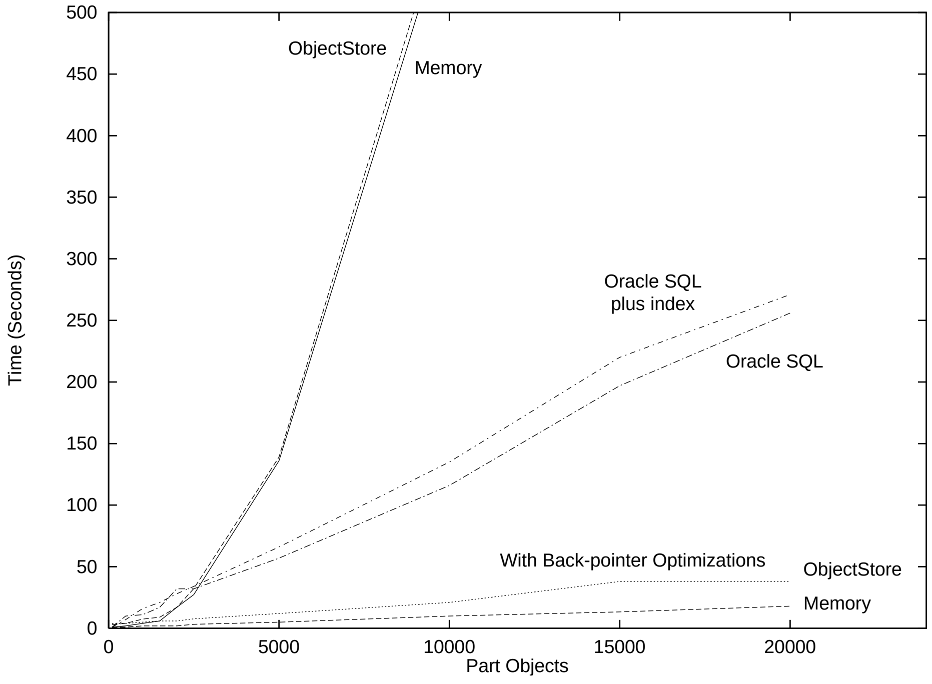

Figure 6.2 — PartStone Timings, Detail Showing Benchmark Results Under 500 Seconds

6.3 BOMStone Results

The BOMStone benchmark was implemented using each of the three SDAI systems. Timing measurements were made for each system using a series of data sets containing between 100 and 20,000 part objects. In addition to the basic algorithm, we tested optimizations based on the unique characteristics of each system. The optimizations are discussed in Section 6.3.3. The BOMStone timings are shown Figure 6.3 and Figure 6.4.

6.3.1 Bill of Material Test Data

The *mkbom* program was developed to generate bill of material data with assemblies of arbitrary size. The generated assemblies were composed of *product definition* objects sewn together by *next assembly usage occurrences*. Each *product definition* was also linked to a unique *product definition formation* and *product* object.

The generated assembly structures were trees. They did not contain any repeated or shared components. As described in Section 5.4.4, this allows us to express the complexity of the BOMStone algorithm in terms of the number of parts.

For the measurements in this work, the generated data sets contained assemblies that were 4 items wide and 6 levels deep for 4096 parts per tree. Ten BOMStone data sets were used. The sizes were: 100, 500, 1000, 1500, 2000, 2500, 5000, 10000, 15000, and 20000 part objects.

6.3.2 BOMStone Timings

The basic BOMStone algorithm was tested against the Oracle, ObjectStore and working-form implementations. The Oracle timings were performed with and without extra indices on the most-used columns.

The ObjectStore and working-form implementations had similar performance, but not as close as with PartStone. The times for both systems showed the growth expected from an O(N^2) algorithm. As with PartStone, the Oracle implementation was significantly slower, and we were unable to complete measurements on data sets larger than 1000 parts. Indices improved the Oracle behavior slightly, but not enough to complete all measurements. The extreme behavior of the Oracle implementation suggests higher degree polynomial behavior.

The Oracle indices were added on keys of the *product definition*, *product definition relationship*, and *assembly component usage* tables, as well as the *relating* and *related product definition* attributes of *product definition relationship*.

6.3.3 BOMStone Optimizations

Additional timings explored the effect of optimizations. As with the PartStone benchmark, most optimizations involved replacing a traversal with efficient Usedin() functionality. In this case, the FindTopLevel() and FindAssembly() functions were both modified to perform a Usedin() operation as efficiently as possible.

Relational Optimizations

The Oracle implementation was modified to perform the FindTopLevel() and FindAssembly() operations using SQL joins. In addition, other SDAI calls were collapsed into single SQL queries where possible.

The modifications improved performance, but we were still unable to complete measurements for all data sets until extra indices were added to the tables. Without extra indices, the modified benchmark required almost a full day to process 10,000 parts, but with indices this time went down to about three minutes.

OODBMS and Memory Optimizations

To improve the FindTopLevel() and FindAssembly() operations, the C++ class for *product definition* was extended to keep back-pointers for the *relating* and *related product definition* attributes of *assembly component usage*. The back-pointers were initialized by traversing all *assembly component usages* at the beginning of the benchmark run. The modified algorithm is shown below:

```
void BOMStone_backpointer (dataset : S)
begin
    foreach assembly_component_usage : acu in S do
    begin
        AppendRelatedPdefBackpointers  (GetRelatedProductDefinition (acu),acu)
        AppendRelatingPdefBackpointers (GetRelatingProductDefinition (acu),acu)
    end

    foreach product_definition : pdef in S do
    begin
        if Empty (GetRelatedPdefBackpointers(pdef))
        then FindAssembly (pdef, 0);
    end
end

void FindAssembly_backpointer (product_definition  pdef, integer depth)
begin
    PrintIndent (depth)     /* indent according to depth */
    PrintProductDef (pdef)  /* print product and version name */

    foreach assembly_component_usage : acu in
        GetRelatingPdefBackpointers (pdef) do
    begin
        FindAssembly (GetRelatedProductDefinition (acu), depth+1)
    end
end
```

This optimization reduces the complexity of the BOMStone algorithm to linear time. Initialization of the back-pointers requires O(A) time, but we can find the owners and components of a part in constant time. Over the course of all assemblies, the FindAssembly() function will touch each part once, requiring O(P) operations. The complete algorithm is O(A+P), but since A is O(P), the complexity becomes O(P+P), or just O(P).

The modifications had a large effect on the observed benchmark times. As predicted, the times for the ObjectStore and working-form implementations grew in a linear fashion. As with the basic algorithm, the ObjectStore implementation was slower than the working-form and grew at a slightly larger rate.

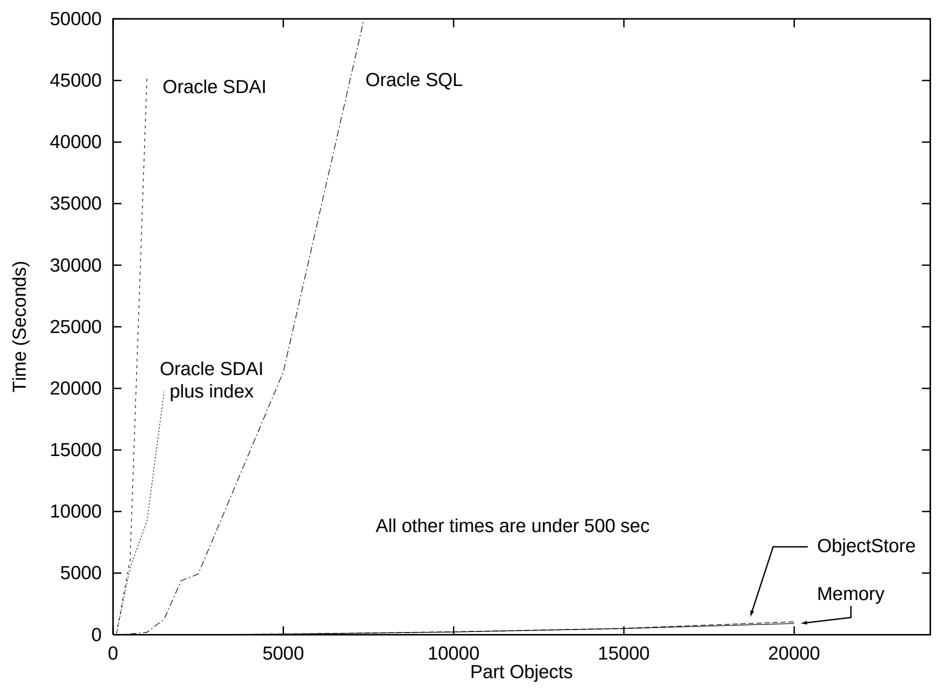

Figure 6.3 — BOMStone Timings

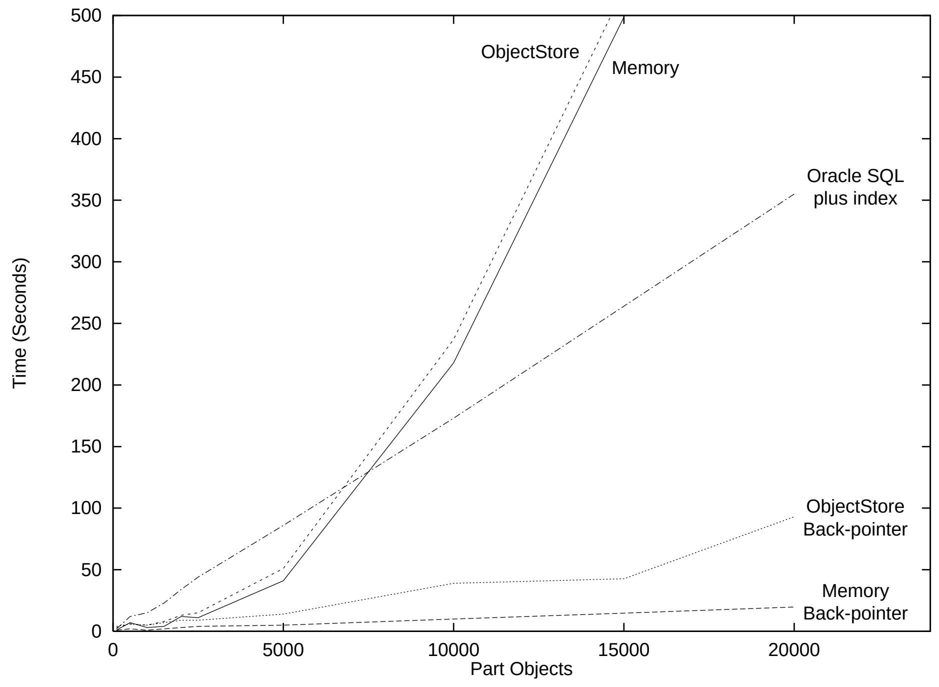

Figure 6.4 — BOMStone Timings, Detail Showing Benchmark Results Under 500 Seconds

6.4 NURBStone Results

The NURBStone tests used a series of data sets containing between 2300 and 100,000 geometric objects. In addition to the basic NURBStone algorithm, several modifications to the Oracle implementation were tested. These optimizations are discussed in Section 6.3.3. The results of all NURBStone timings are shown Figure 6.6 and Figure 6.7.

6.4.1 Shape Test Data

Creation of physically precise shape data from scratch would require the services of a CAD system or geometric modeling kernel. However, the complexity of the NURBStone benchmark does not depend on the composition or numeric accuracy of individual shape representations. The *mknurb* program was developed to create data sets of varying size by replicating an existing shape data set.

The “moon buggy” test part shown in Figure 6.5 was replicated as needed to create data sets of sufficient size. The moon buggy data set was created for the STEPnet inter-operability testing efforts [Down96] and has been used by over a dozen CAD vendors to ensure that their software conforms to the standard. The test part has 20 shape representations and uses most of the AP-203 geometric entity types. The data sets were constructed in a range of sizes between 2300 and 100,000 geometry objects. The number of “moon buggy” copies and exact sizes of the data sets are shown in Table 6.1.

6.4.2 NURBStone Timings

As with the previous benchmarks, the basic algorithm was tested against each SDAI implementation. The Oracle timings were performed with and without extra indices on the most-used columns.

Generated Data Set	Copies	Total Instances
ndata-002-3	1	2343
ndata-004-6	2	4686
ndata-007-0	3	7029
ndata-009-3	4	9372
ndata-011-7	5	11715
ndata-014-0	6	14058
ndata-021-0	9	21087
ndata-051-5	22	51546
ndata-100-0	43	100749

Table 6.1 — NURBStone Data Set Sizes

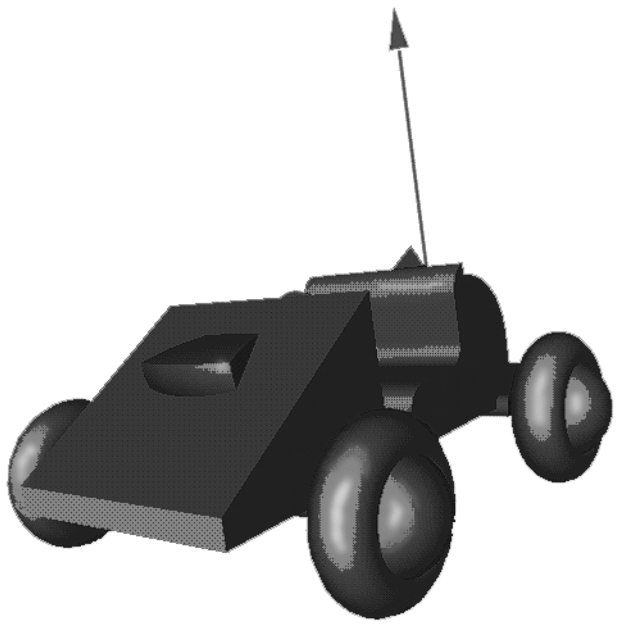

Figure 6.5 — STEPnet Moon Buggy

The working-form times were fastest and show linear growth. The ObjectStore implementation was slower and differed from the working-form by a larger margin than with the other benchmarks. The ObjectStore times are somewhat linear, but there is a suggestion of non-linear behavior at larger data set sizes.

As with the other benchmarks, the Oracle times were much slower, although we were able to complete measurements on all data set sizes. The Oracle times show a definite non-linear trend. Addition of indices resulted large reduction in the times and return to linear behavior. The Oracle indices were added to the keys of all geometric entity tables.

6.4.3 NURBStone Optimizations

Two additional timings runs explored modifications to the Oracle implementation. The Oracle NURBStone was modified to collapse multiple SDAI calls into single SQL queries where possible. The modifications were tested with and without additional indices.

The modifications had minimal effect on the performance of the benchmark. In both timings, the modified algorithm was only slightly faster than the original. The benchmark was more strongly affected by the presence or absence of indices than by the clustering of attribute access operations.

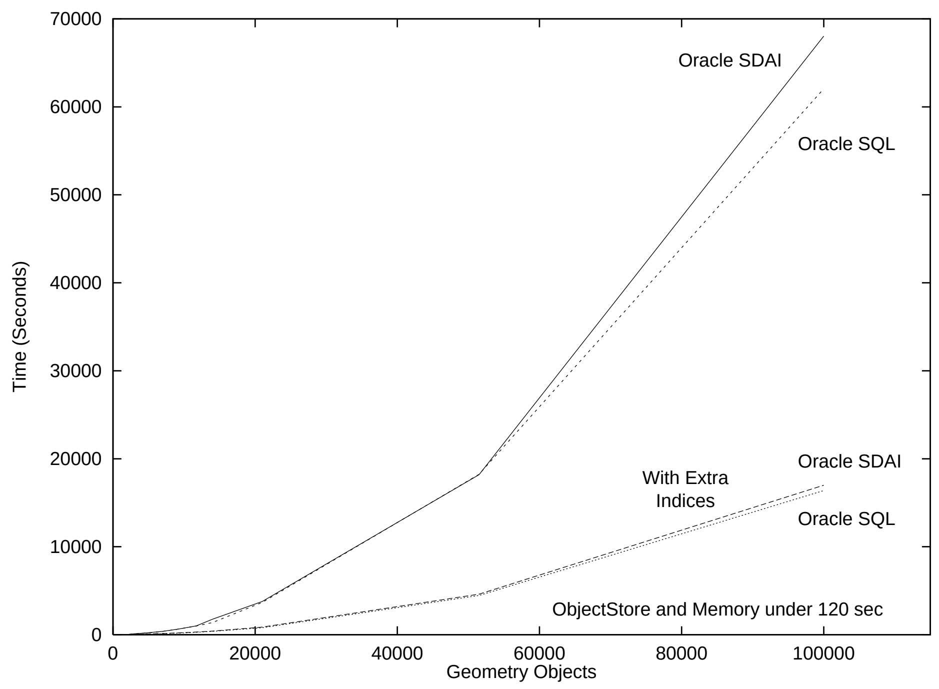

Figure 6.6 — NURBStone Timings

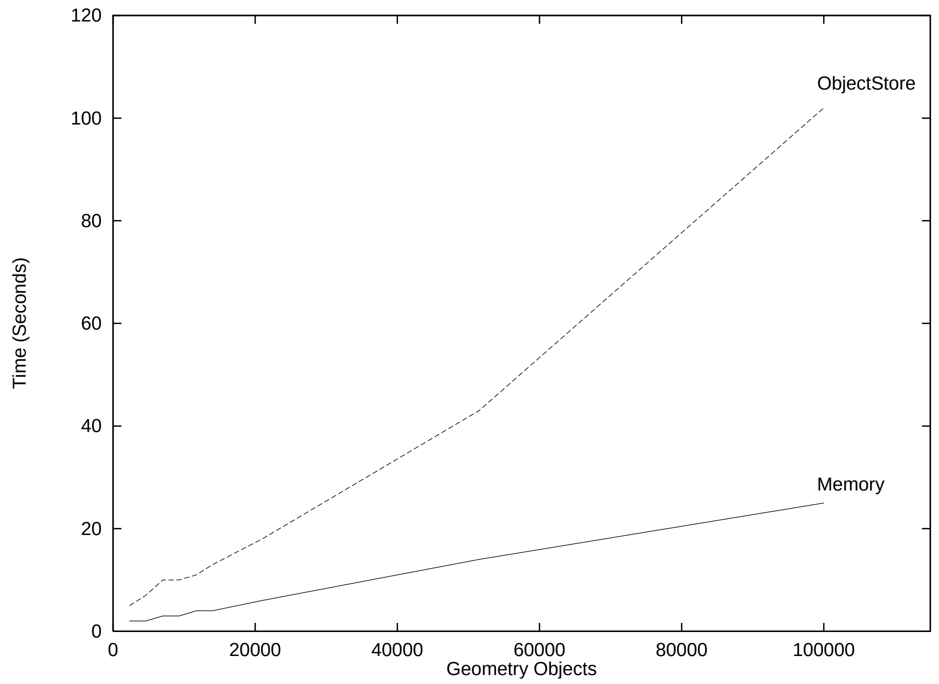

Figure 6.7 — NURBStone Timings, Detail Showing Benchmark Results Under 120 Seconds

6.5 Database Load/Unload Results

In order to gain insight into the performance of upload/download and cached bindings, we measured the times required to load and extract data sets from Oracle and ObjectStore. The measurements reflect the time required to load the database from a physical file as well as the time required to extract data from the database to a physical file. In Section 7.5, we combine these numbers with the results of the working-set benchmarks to estimate the performance of alternate bindings.

Figure 6.8 and Figure 6.9 show load and extract times for the NURBStone data sets. Oracle requires much more time to transfer data than does ObjectStore. Aside from the difference in magnitude, extracting data from Oracle appears to be more costly than loading it, while for ObjectStore, the opposite is true.

In Section 3.5.1, we note that the load and extract programs should behave in O(N) fashion. The load behavior for Oracle is strongly linear, while extraction shows higher-order behavior. In fact, we were unable to complete all of the measurements because of the magnitude of the extract times. Addition of indices improved the extract times slightly, but did not change their fundamental behavior. The ObjectStore extract behavior was linear while the load behavior showed signs of higher complexity.

We repeated the load and extract tests using PartStone and BOMStone data sets. The ObjectStore results were identical. The Oracle results were identical except for the case where indices were applied. Indices reduced extract times to almost the same values as the load times. This was unexpected because indices did not strongly affect the behavior of NURBStone data sets. This suggests that either the extract software or Oracle indices may depend on the number of entity types in the database. The PartStone and BOMStone data sets have a small number of entity types (two and four respectively), while the NURBStone data sets use over fifty different types.

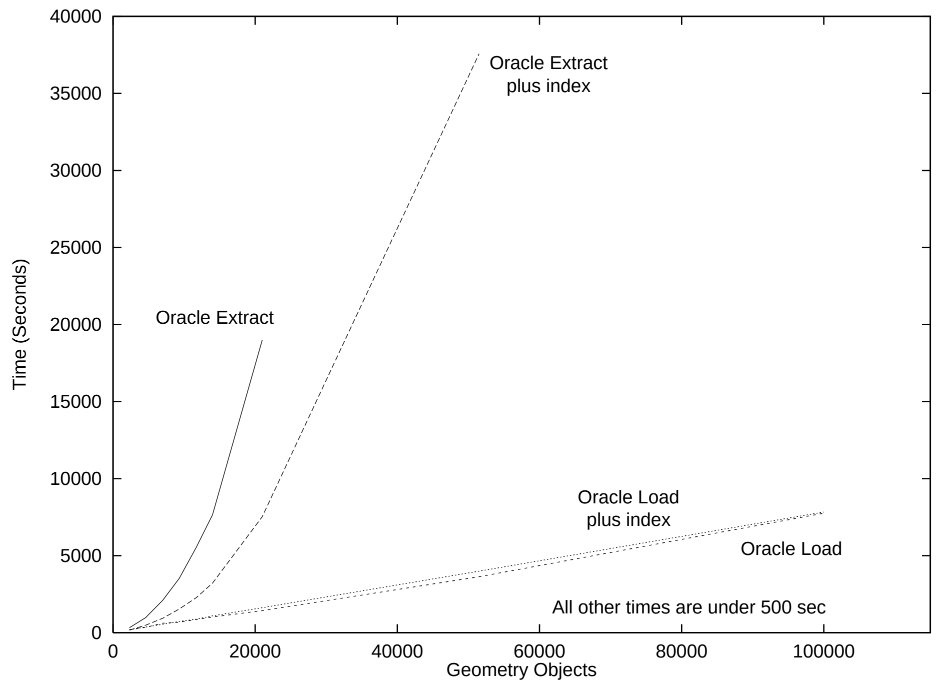

Figure 6.8 — Database Transfer Times, Computed Using the NURBStone Data

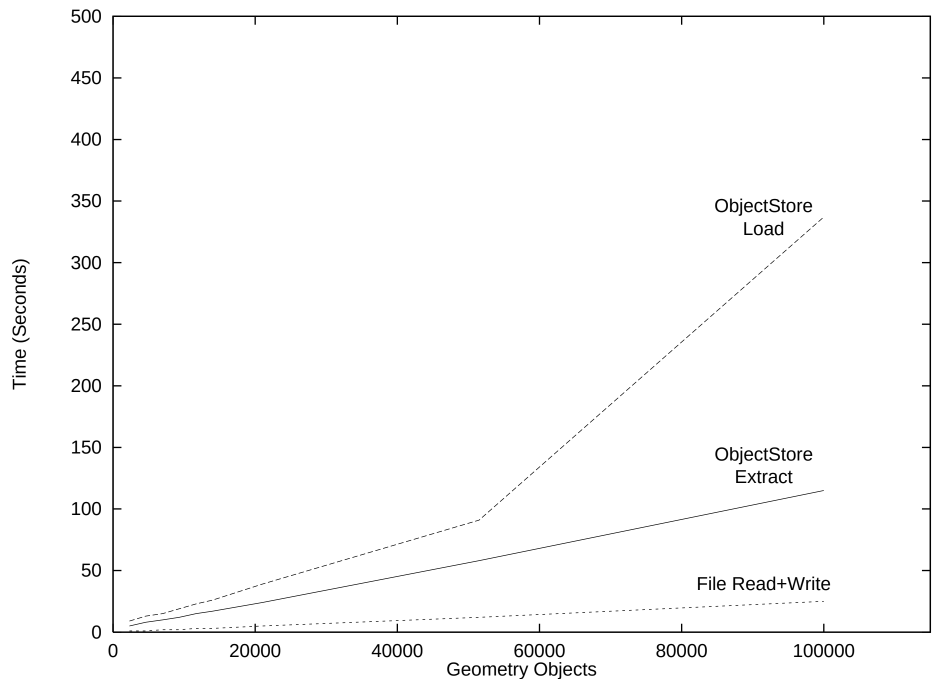

Figure 6.9 — Database Transfer Times, Detail Showing Times Under 500 Seconds

7 Discussion

7.1 Overview

We have defined several benchmarks for STEP database implementations and have tested them against SDAI implementations built on top of Oracle, ObjectStore and a main-memory working form. These results have given us some insight into four factors that affect the performance of an SDAI implementation.

7.2 Effect of Access Performance

In Section 5.5.4, we determined that the NURBStone benchmark was an O(N) algorithm. Looking at the benchmark results, we expect to see linear behavior. This is true for the working-form binding, but we see slightly nonlinear behavior with ObjectStore and definite nonlinear behavior with Oracle.

The benchmark complexity analysis assumed that attribute access was a constant-time operation. The benchmark results have shown that this assumption is not always valid. Using the NURBStone results, we can compute the time required per object. This time is related to the cost of access for each system. It is important to note that these are relative

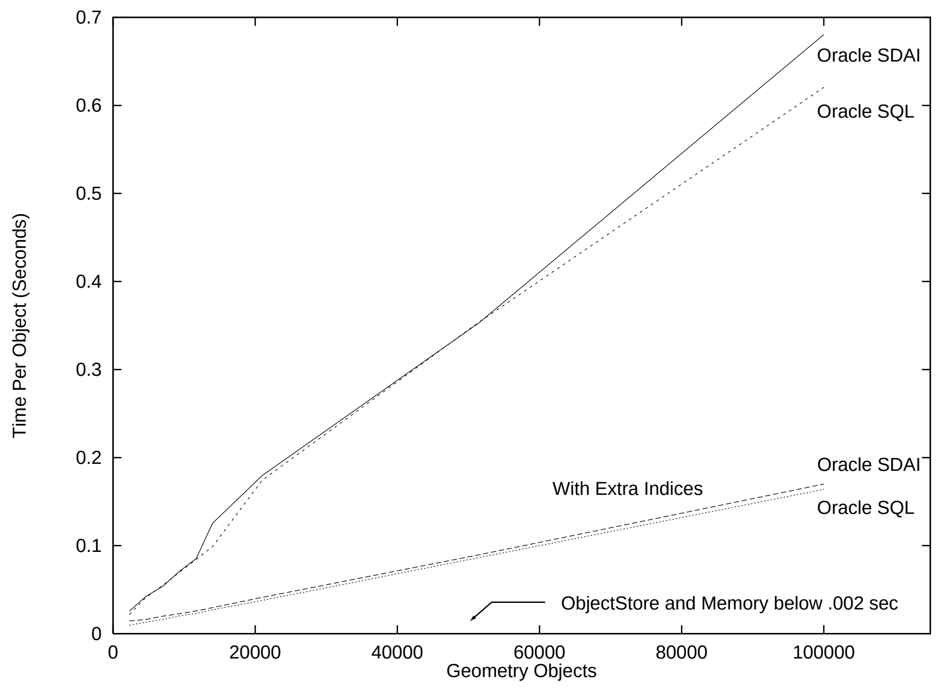

Figure 7.1 — Average Access Time per Object, Computed Using the NURBStone Benchmark Results

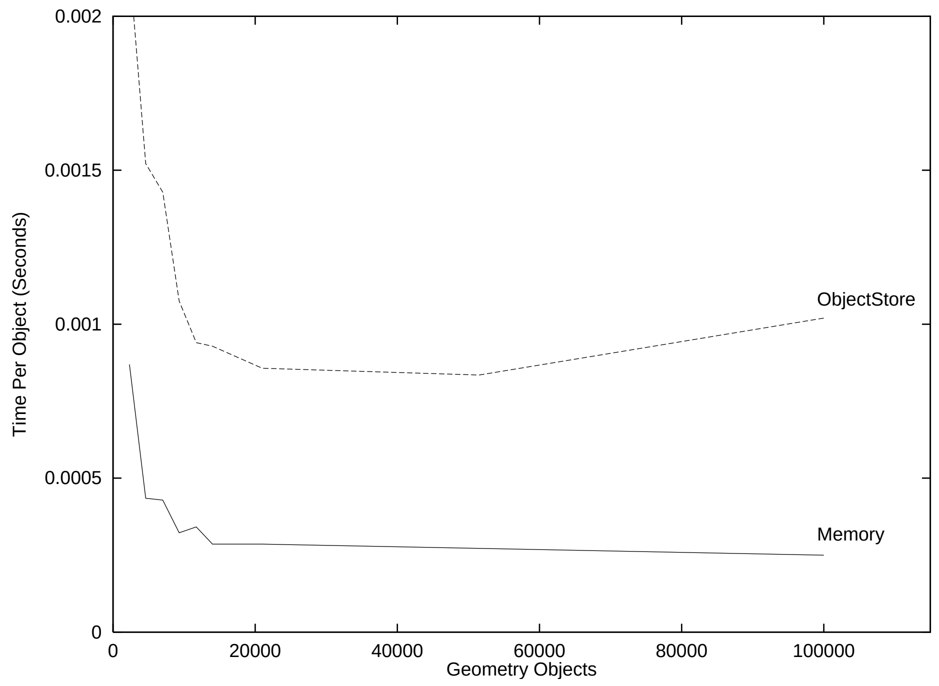

Figure 7.2 — Average Access Time per Object, Detail Showing the ObjectStore and Working-Form Values

costs. The exact cost of access will vary based on the mix of instructions, but should be in line with the results presented here.

Figure 7.1 and Figure 7.2 show the relative costs of access for each system. The working-form has the lowest cost. The ObjectStore cost is several times as large, and there is some indication that the cost may be non-constant. The Oracle implementation has a cost several orders of magnitude larger that grows linearly with the size of the database.

System	Cost per Object	Objects per Second
Oracle	~.05-.7 sec/obj	1.4 - 20 obj/sec (depends on indices and DB size)
ObjectStore	~.001 sec/obj	1000 obj/sec
Working-Form	~.00025 sec/obj	4000 obj/sec

Table 7.1 — Database Access Costs

A variable cost of access increases the complexity of the benchmarks. As discussed in Section 5.3.4, the complexity of the basic PartStone and BOMStone algorithms is $O(N^2)$. Assuming each relational access behaves as $O(N)$, the algorithms require $O(N^2)$ accesses at $O(N)$ each, or $O(N^3)$ behavior. This corresponds with the extreme behavior of the relational implementations in Figure 6.1 and Figure 6.3.

According to these results, we would expect the database extract utility to behave in an $O(N^2)$ fashion. This appears to be consistent with most of the extract times in Section 6.5, but not with PartStone and BOMStone extract times on heavily indexed databases. Additional investigation is needed on the effect of indices and differing numbers of entity types on access costs.

7.3 Effect of Usedin() Optimizations

The PartStone and BOMStone benchmarks required a Usedin() operation to traverse an existence-dependent relationship. This type of relationship is common in the STEP models and is often used to model associated properties.

These experiments showed the importance of a well designed SDAI Usedin() algorithm. In the case of the relational system, use of SQL joins reduced higher-order behavior to linear time. Rewriting the ObjectStore and working-form implementations to use back-pointers had a similar effect.

The PartStone results (Figure 6.2) show that even with a much lower cost of access, the non-back-pointer ObjectStore and working-form systems were not able to compete with an Oracle query after 2500 objects.

These optimizations were hand-coded for maximal efficiency. A general purpose SDAI Usedin() implementation may not be able to reach the same performance as hand optimized code. In any case, the benchmark results show this to be a more important factor than access speed for an SDAI binding that must work on the types of product data characterized by the PartStone and BOMStone benchmarks.

The back-pointer implementations were faster than SQL joins, but they require the proper data set to be selected a-priori. In a very large SDAI database, an SQL join might be more useful for selecting the proper data sets for later use with a cache. This technique is discussed in [Sama90].

7.4 Effect of Relational Index Optimizations

The indices did not have as large an effect as expected. Extra indices improved times slightly in most cases. They had a large effect on the optimized BOMStone benchmark, a moderate effect on the NURBStone, but only marginal effect on the PartStone.

	SDAI	**SQL**
PartStone	Slightly Better	Slightly Worse
BomStone	Slightly Better	Much Better
NURBStone	Better	Better

Table 7.2 — Effect of Indexing on Benchmarks

In general, indexing seems to be most useful in improving the SQL joins of Usedin() optimizations. It also improves SQL select behavior, such as that in SDAI attribute access, but does not reduce access cost to a constant time.

The extract operations showed varied behavior in the presence of indices. As noted in Section 6.5, indices were most beneficial for PartStone and BOMStone data, but did not strongly affect times for NURBStone data. More investigation is needed to determine the exact nature of this behavior.

7.5 Effect of Access Architecture

In the course of this work, we gathered benchmark performance statistics for direct SDAI bindings on Oracle and ObjectStore, as well measurements of database load and extract times. We can combine the load times from Section 6.5 with the working-form benchmark times to estimate the behavior of upload/download and cached bindings.

These estimates are accurate for upload/download bindings, but under-estimate the performance of cached bindings, which do not require a file round trip. The cost of a file round trip is small (see Figure 7.2), but as noted in Section 4.4.1, an Oracle cached binding was constructed and could be used if exact times are required.

7.5.1 ObjectStore Alternate Bindings

The alternate bindings compare very well to the ObjectStore direct binding. Figure 7.3 shows that, for the basic PartStone and BOMStone algorithms, the bindings are almost indistinguishable.

At smaller times, the cost of the file round trip becomes large compared to the benchmark. The bindings remain close with the optimized algorithms in Figure 7.4, although the direct binding is slightly faster. In Figure 7.5, with the NURBStone benchmark, the direct binding is faster, but the alternate bindings still have reasonable performance.

These results indicate that an ObjectStore alternate binding would have similar performance characteristics to the direct binding, with reduced database functionality, but also reduced implementation cost. Assuming that the ObjectStore and Versant database systems have similar performance characteristics, these findings validate the cost-effectiveness of the Versant cached binding implementation in Section 4.4.2.

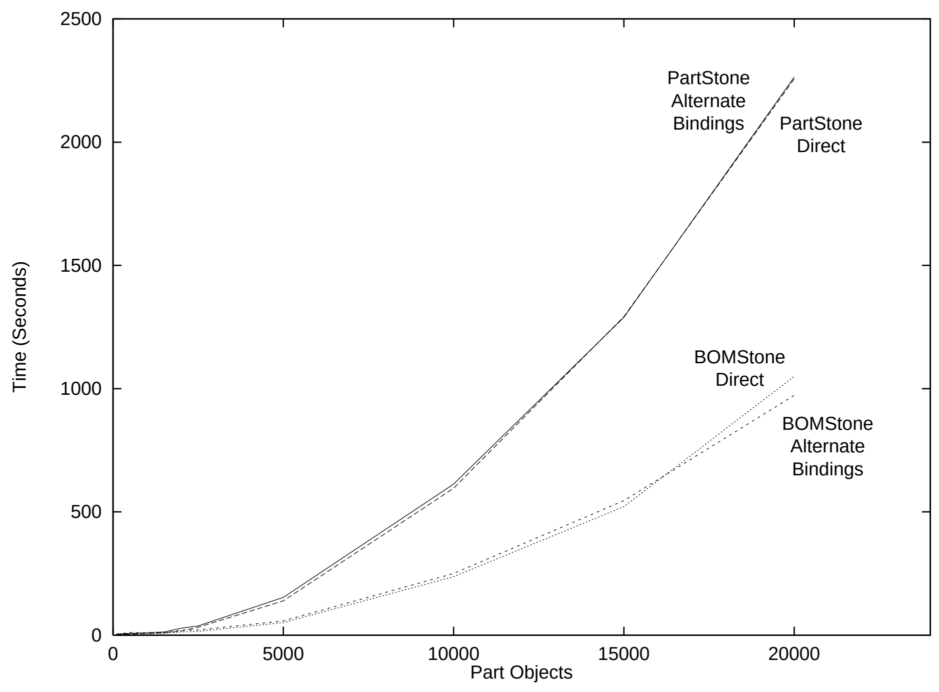

Figure 7.3 — ObjectStore Alternate vs. Direct Bindings, Basic PartStone and BOMStone Algorithms

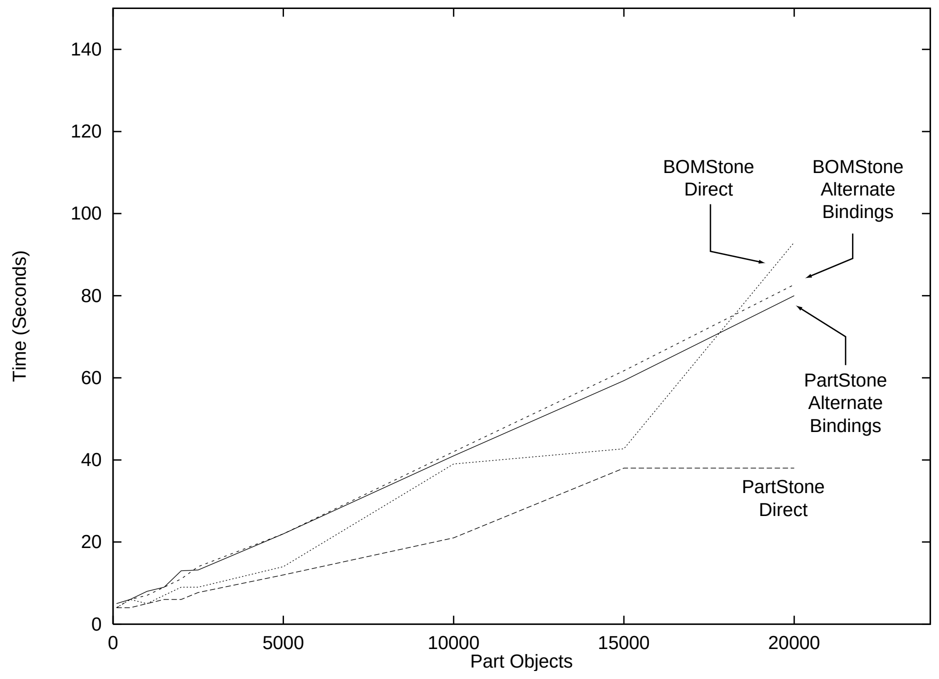

Figure 7.4 — ObjectStore Alternate vs. Direct Bindings, Optimized PartStone and BOMStone Algorithms

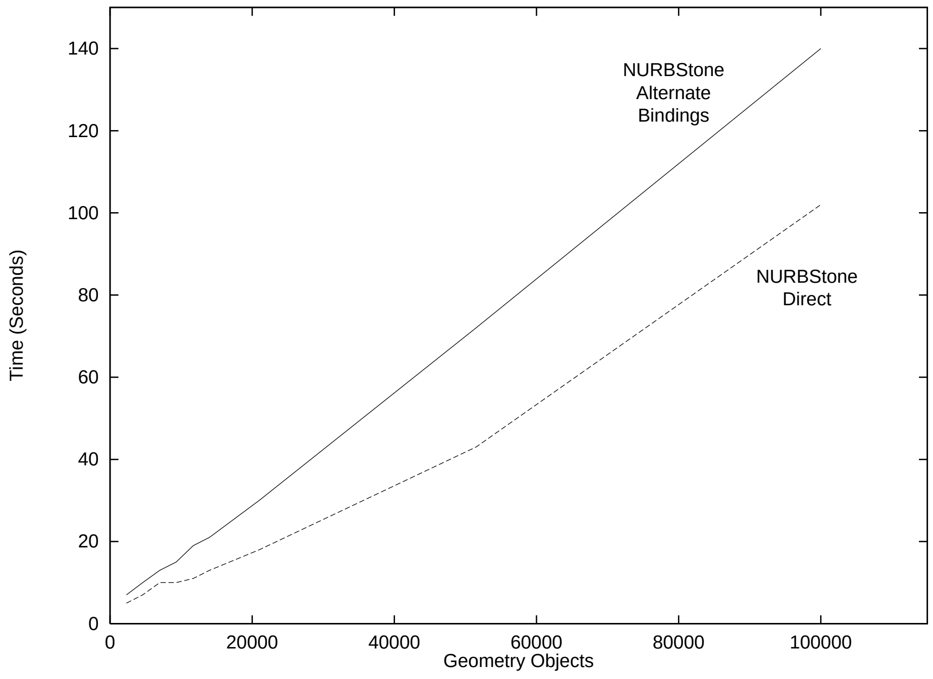

Figure 7.5 — ObjectStore Alternate vs. Direct Bindings, NURBStone Benchmark

7.5.2 Oracle Alternate Bindings

The alternate binding estimates were constructed with best-case times. The estimates used extract times from indexed databases and working-form times with back-pointer Used-in() optimizations.

On the PartStone (Figure 7.6) and BOMStone (Figure 7.7) benchmarks, the alternate bindings were significantly faster than the direct binding basic SDAI algorithms, but they were not faster than direct binding optimized algorithms. On the NURBStone benchmark (Figure 7.8), the alternate bindings were slower than the direct binding in all cases.

In Section 7.2 we saw that the cost of an Oracle access operation is several orders of magnitude slower than an ObjectStore access. This extra cost affects all Oracle bindings, but in different ways. Oracle upload/download and cached bindings have extremely long latencies. The upload/download binding can side-step the latency by pre-fetching application data.

An Oracle direct binding incurs a high cost for each SDAI call, but programs can take advantage of SQL operations to reduce the number of calls. If an application can use SQL queries to do a large amount of work in a single call, a direct binding makes the most operational sense. If applications must perform a large number of SDAI calls, particularly with higher-order complexity algorithms, one of the alternate bindings would provide better performance.

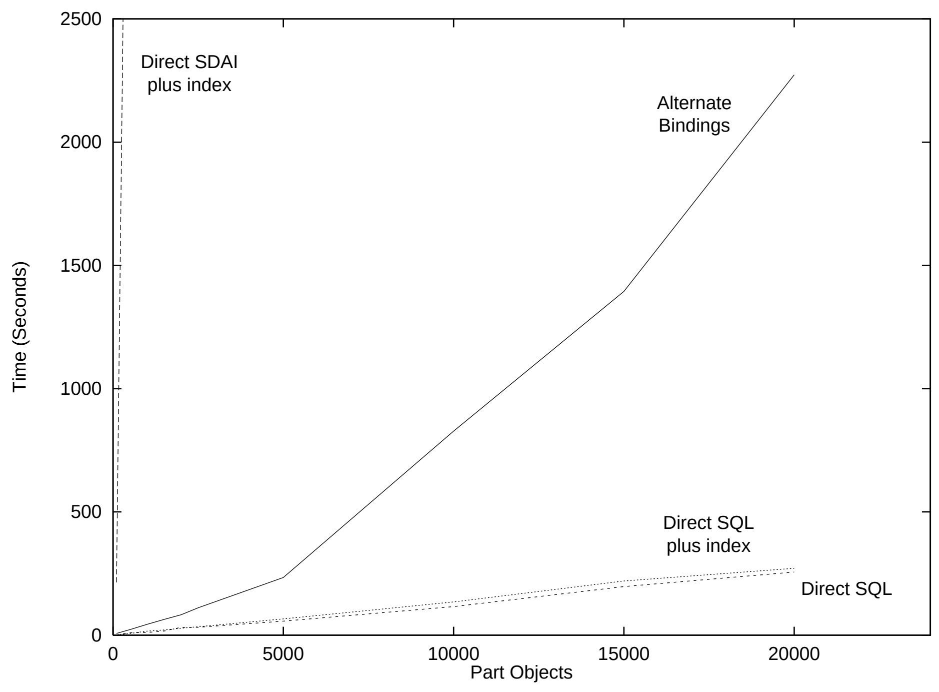

Figure 7.6 — Oracle Alternate vs. Direct Bindings, PartStone Benchmark

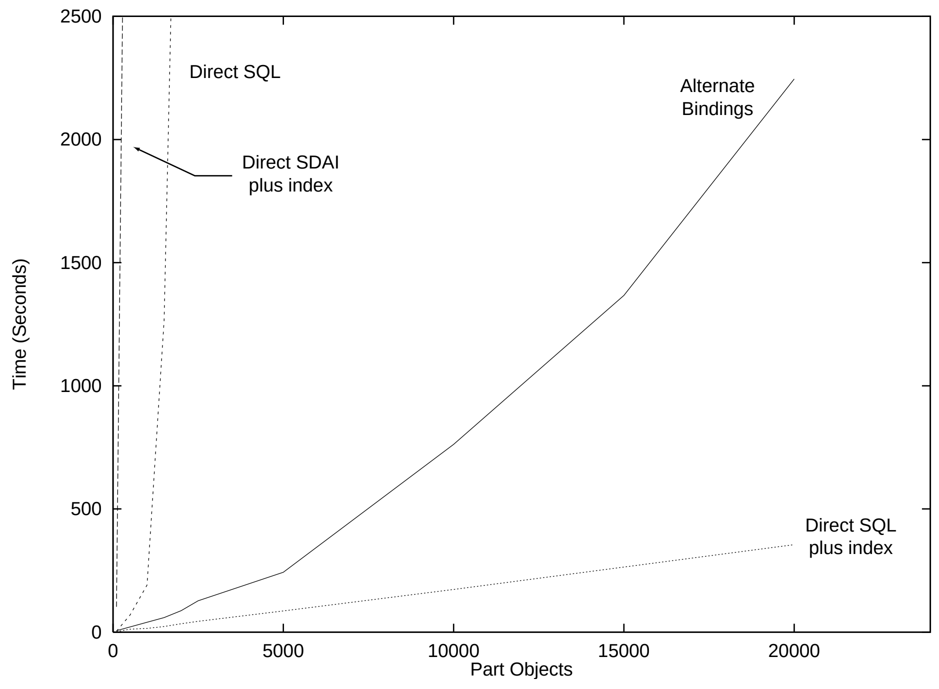

Figure 7.7 — Oracle Alternate vs. Direct Bindings, BOMStone Benchmark

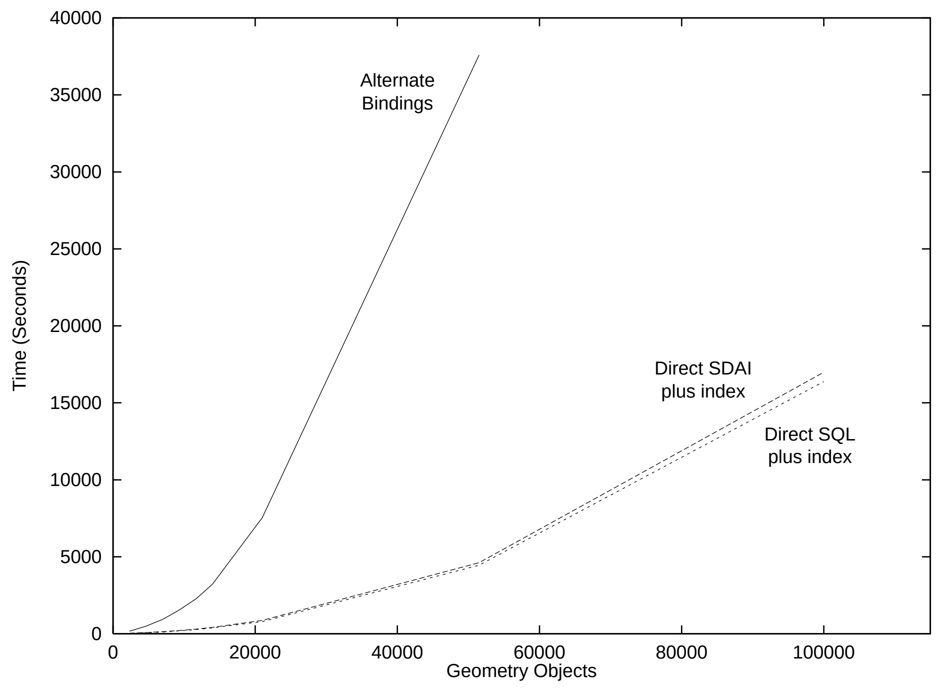

Figure 7.8 — Oracle Alternate vs. Direct Bindings, NURBStone Benchmark

8 Conclusions

8.1 Summary

Product databases based on EXPRESS models can reduce industry's dependence on vertically integrated engineering applications and proprietary product databases. We have performed the following work to simplify and promote construction of product databases:

- Identified several design decisions affecting the structure of EXPRESS databases and SDAI database bindings.
- Examined the effect of binding architecture on implementation cost by looking at six prototype SDAI database bindings.
- Identified a set of benchmarks for measuring the operational characteristics of an SDAI binding on a STEP AP-203 database.
- Used these benchmarks to measure the baseline performance of implementations built on Oracle and ObjectStore, as well as the effect of several optimizations.
- Examined the effect of binding architecture on operational cost by comparing the performance measurements for direct SDAI bindings on Oracle and ObjectStore with values calculated for the alternate bindings.

In the following sections, we summarize the conclusions and contributions of this research and discuss areas that might benefit from future exploration.

8.1.1 Implementation Framework

We identified two important architecture decisions for SDAI database bindings. The first factor is data access style. The simplest style is an upload/download binding, which consists of off-line file upload/download tools paired with a working-form SDAI binding. Next is a cached binding, which moves data to and from a main-memory cache. The final style is a direct binding, which manipulates data "in place" by calling one or more native operations for each SDAI operation.

The second factor that an implementor must consider is whether to use code generation or data-dictionary access. This choice depends only on features of the underlying database and the need for customization of the SDAI binding. The architecture factors are summarized in Figure 8.1.

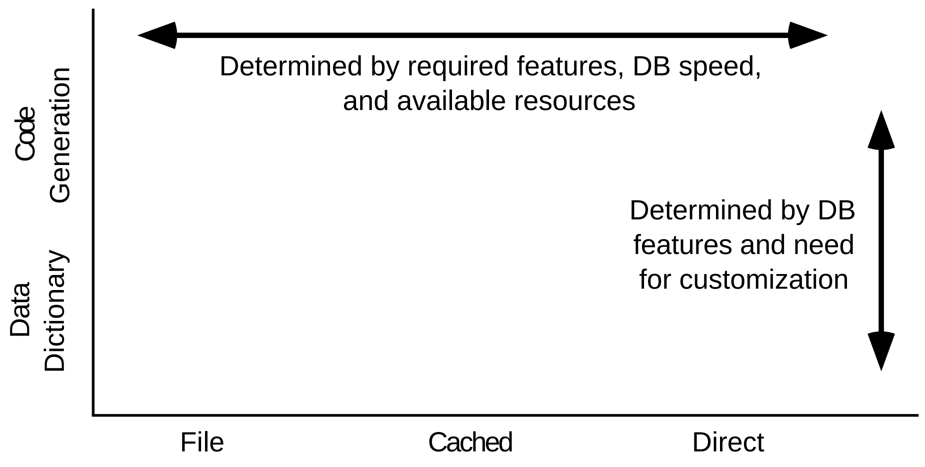
Determined by required features, DB speed, and available resources
Determined by DB features and need for customization
Code Generation
Data Dictionary
File
Cached
Direct

Figure 8.1 — SDAI Binding Design Decisions

We examined the SDAI implementations covering the architectures shown in Table 8.1 and noted that the cost for implementing each access style rises with the number of features it provides. The upload/download and cached bindings provide SDAI access, but are not useful for situations requiring concurrent update. A direct binding is more costly, but can make use of more database features and optimizations.

Oracle and OpenODB Early-Bound Upload/Download	Oracle Early-Bound Cached	ObjectStore and Oracle Early-Bound Direct
Late-Bound Upload/Download	Versant Late-Bound Cached	Late-Bound Direct

Table 8.1 — SDAI Architectures Covered by the Implementation Studies

We conclude that, if the application requirements permit, the most cost-effective implementation approach is to use code generation to construct upload/download software. Once testing and development are complete, the load and extract programs can be integrated with the working-form SDAI to produce a cached binding. This architecture permits interfaces for several systems to be integrated with the binding, as shown in Figure 8.2.

Experience from these projects shows that code generation requires relatively minor changes to existing EXPRESS compiler software, and the limited scope of upload/download software makes it far simpler than a direct SDAI binding. Furthermore, the cost of developing and maintaining a working-form SDAI binding can be distributed across several projects.

Finally, by keeping the database access software simple and separate from the SDAI binding, future systems can explore the benefits of a client-server model with network-wide distribution of data. This is particularly true now that more applications are moving towards web-centric designs and a Java SDAI specification is under development. If engineering data must be distributed to thin clients, a cached binding becomes very attractive.

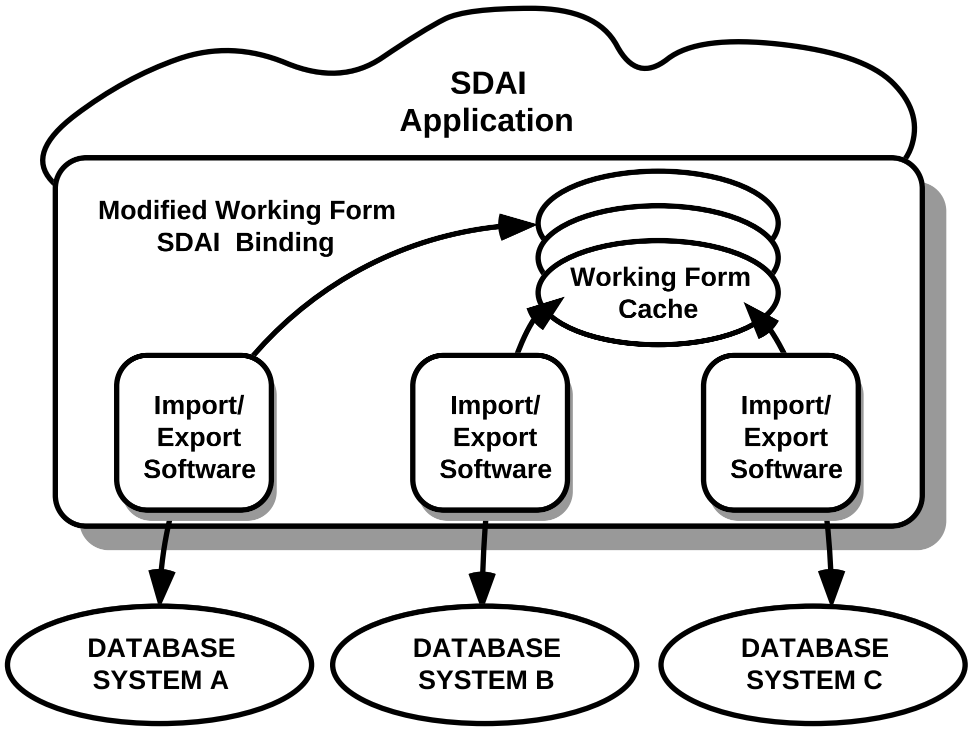

Figure 8.2 — Preferred Implementation Architecture

A direct binding would require complex client software and network communication for each SDAI operation, but a cached binding could keep database load and unload software on the server and transfer data in one burst to a lightweight Java working-form client.

8.1.2 SDAI Benchmarks

We identified a set of benchmarks to measure the operational characteristics of SDAI implementations. These benchmarks are based on the STEP AP-203 information model, but can be applied to most STEP information models. The PartStone benchmark operates on part version information, which is modeled in an existence-dependent style. The NURB-Stone benchmark operates on product shape information, which is modeled in a navigational style. Finally, the BOMStone benchmark operates on bills of material, which is modeled in a mix of the two styles.

We used these benchmarks to evaluate the performance of ObjectStore and Oracle direct bindings. In addition, we tested a working-form SDAI binding and measured the load and extract performance of the database systems. From this information, we were able to determine the performance of upload/download and cached bindings. These results are summarized in Table 8.2.

	Upload/Download	**Cached**	**Direct**
ObjectStore	Fast	Fast	Slightly Faster
Oracle	Slow, but can be set up beforehand for interactive code.	Slow. Better than direct for complex algorithms w/o SQL optimization.	Slow. Better speed if SQL optimizations are available.

Table 8.2 — Performance of SDAI Bindings

These measurements showed that the access and update performance of Oracle and ObjectStore behave differently as the database size increases, which may influence the desirability of each binding. For the ObjectStore system, we saw that upload/download and cached bindings perform just as well as a direct binding. This performance, coupled with the low implementation cost, reinforces the desirability of a cached binding.

For the Oracle system, we saw that the cost of an access operation is several orders of magnitude slower than with ObjectStore. This severely impacts the speed of all bindings, but in different ways. The upload/download and cached bindings result in extremely long latencies, but the upload/download binding can lessen this by pre-fetching the application data. The direct binding will incur a high cost for each SDAI call, but programs can take advantage of SQL operations to reduce the number of calls. If an application can use SQL queries to do a large amount of work in a single call, a direct binding makes the most operational sense. If applications must perform a large number of SDAI calls, particularly with higher-order complexity algorithms, then one of the alternate bindings would provide better performance.

We also tested the effect of optimizations on the underlying database. In particular, we note the importance of Usedin() optimizations to any SDAI implementation. For the Oracle implementation, experience shows that the addition of indices for SDAI attribute access does not have as large an effect as the addition of indices for SQL joins and Usedin() optimizations.

8.1.3 Recommendations for Implementors

The results of this research show that implementors must examine the application requirements for the product database. If access to data at model granularity is sufficient for applications, one of the alternate binding architectures (cached or upload/download) should be considered because of the lower implementation cost. If applications require low latency for access to individual data values, a direct binding should be used, even though it has a greater implementation cost.

The SDAI session architecture provides for access to data stored in multiple repositories (database systems). Simultaneous access to different database systems will usually require an alternate binding, as shown in Figure 8.2, due to the complexity of adding access code for multiple databases to each SDAI operation. However, a common access protocol, like ODBC, might allow a direct binding to be used with more than one database system.

The complexity of applications also affects the choice of binding architecture. In general, SDAI applications that have greater than linear complexity, such as the PartStone and BOMStone tests, will perform as well or better with the low implementation cost alternate binding architectures. This is because database operations are slower than memory access. The alternate bindings extract data from the database using a linear number of database operations, and then perform SDAI algorithm operations at main-memory speeds. A direct binding must perform all of the high-complexity algorithm operations at the slower database speeds.

Applications that use linear time SDAI algorithms, like the NURBStone benchmark, may perform slightly better with a direct binding, although this requires greater implementation effort. Also, higher-order algorithms can be affected by database optimizations. In particular, optimizations on the Usedin() operation should be given a high priority. These offer the potential for large performance gains on algorithms that traverse existence-dependent information.

8.2 Contributions

The main purpose behind the SDAI is to reduce the cost of engineering applications. Industry needs the SDAI to make applications portable across different storage technologies, and to encourage the development of product databases. Until now, there has been no work in the field that discusses how to build an SDAI implementation, or how to anticipate what the costs of an implementation will be. This work offers guidance through the following contributions:

- Definition of a framework for database implementation of EXPRESS models. These implementation architectures can be grouped into upload/download, cached, and direct implementations. The implementation cost for the architectures have been illustrated using systems built on a variety of databases.

- Definition of a representative set of benchmarks for evaluating the operational costs of database implementations. The PartStone, BOMStone, and NURBStone benchmarks were developed using AP-203, but the definitions they are based on are shared by many of the STEP application protocols.

- Measurements of SDAI binding operational characteristics on database systems that are commonly used by engineering applications.
- Recommendations for implementors based on application requirements and the relative costs of implementation and operation for each implementation architecture.

These contributions should simplify construction of EXPRESS database implementations by providing a well-defined framework and examples. In addition, the results presented in this work should improve the quality of implementations by ensuring design decisions appropriate to the intended use of the system. In the larger view, it is hoped that these contributions will help industry to integrate design and manufacturing processes and reap the benefits of concurrent engineering.

8.3 Future Work

In the course of this work, we have raised some questions that would benefit from additional investigation. For example, results described in Section 6.5 suggest that Oracle extract software or Oracle indices might depend on the number of entity types in a database. Additional experiments could be run to determine the exact nature of this behavior.

It would be interesting to investigate the range of algorithms appropriate for implementing each of the three high-level SDAI access architectures. For example, what are the best algorithms for database load and unload programs? How should a direct binding maintain temporary state information? How should SDAI models be integrated back into the database during the upload process?

A number of CAD vendors are considering SDAI bindings to their systems. What would change if the underlying system were not a general-purpose database, but rather an engineering system with its own data structures? An implementor would not use a general

EXPRESS to DDL mapping. Instead, one must deal with a mapping from specific system structures to a specific STEP information model. A direct binding would be similar to a incremental CAD translator. If a direct binding must be constructed, do algorithms exist that enable incremental translation without excessive state information?

The work being done by the ODMG organization could reduce the work needed to implement EXPRESS on object-oriented databases. If the ODMG interfaces are implemented by OODBMS vendors, only one EXPRESS schema mapping and SDAI binding would be needed for compliant systems. However, it should be noted that the ODMG work may not address performance issues raised by this document. Also, it would not be useful for implementations on non-object-oriented systems. Furthermore, the future of ODMG may be in question since the leading OODBMS vendor (Object Design) appears unwilling to continue work in this area.

The OMG standards are another area of interest. As noted in Section 2.3.4, an SDAI binding to the CORBA/IDL language is under development. A CORBA binding could facilitate the development of network accessible product databases by allowing developers to build data servers based on EXPRESS databases [Hard95c]. This is one of the areas currently being explored by the participants in the National Industrial Infrastructure Protocols Consortium (NIIIP).

This consortium is exploring technologies that may reduce costs for concurrent engineering and virtual enterprises. In recent demonstrations, they have used the World Wide Web and CORBA to make STEP product data available to distributed engineering groups.

The Internet, CORBA, and to some extent OLE, are transport technologies that enable wide-area access, but they do not address the meaning of the information that they make available. The World-Wide Web is based on transport technologies but uses the HTML language to describe distributed documents. STEP can play the same role as HTML for CAD data and other technical design information. Combining the international standard for digital product data with the World-Wide Web and vendor-driven software integration technologies such as OMG’s CORBA or Microsoft’s OLE may extend the massive collaboration of the World-Wide Web to design and manufacturing [Loff95].

9 References

[Birc85] E. B. Birchfield and H. H. King, “Product Data Definition Interface (PD-DI),” *Proc. of the 1985 USAF CIM Industry Days*, Texas, April 1985.

[Booc98] G. Booch, J. Rumbaugh, and I. Jacobson, *Unified Modeling Language User Guide*, Addison Wesley, 1998. ISBN 0-201-57168-4.

[Bruc92] T. A. Bruce, *Designing Quality Databases with IDEF1X Information Models*, Dorset House Publishing, New York, 1992.

[Brac85] R. Brachman and H. Levesque, eds., *Readings in Knowledge Representation*, Morgan Kaufman, Los Altos, Calif., 1985.

[CFI92] *Design Representation Programming Interface - Electrical Connectivity*, CAD Framework Initiative, Inc., Austin, Texas, 1992.

[Chen76] P. Chen, “The Entity Relationship model — Towards a Unified View of Data,” *ACM Transactions on Database Systems*, Vol. 1, No. 1, March 1976, pp. 9-36.

[Clar90] S. Clark, “An Introduction to the NIST PDES Toolkit,” Technical Report NISTIR 4336, National Institute of Standards and Technology, Gaithersburg, Maryland, May 1990.

[Curn76] H.J. Curnow and B.A. Wichmann, “A Synthetic Benchmark,” *The Computer Journal*, 19,1 (1976), 43-49. (Also see the comp.benchmarks FAQ for more information about this and many other CPU benchmarks).

[Date86] C. J. Date, *An Introduction to Database Systems*, Fourth Edition, Addison-Wesley, 1986.

[Date89] C. J. Date, *A Guide to the SQL Standard*, Second Edition, Addison-Wesley, 1989.

[Down96] B. Downie, “STEP and SAT: Competing Standards?” *Spatial Relations*, Vol. 13, Spatial Technology, Boulder, Colorodo, April 1996.

[Egge88] J. Eggers, “Implementing EXPRESS in SQL,” Document TC184/SC4/WG1 N292, ISO, Geneva, October 1988.

[Elma89] R. Elmasri and S. Navathe, *Fundamentals of Database Systems*, Benjamin-Cummings, Redwood City, Calif., 1989.

[Gall84] S. Gallagher, *Inside the Personal Computer: A Pop-up Guide*, Cross River Press, 1984. ISBN 0-89659-504-8

[Gadi94] A. Gadient, G. Graves, and J. Boudreaux, “PreAmp: A STEP-Based Concurrent Engineering Environment for Printed Circuit Assemblies,” *Concurrent Engineering: Research and Applications 1994 Conference Proceedings*, August 1994, pp. 529-537.

[Hard91] M. Hardwick et al., “Managing Change Using STEP/EXPRESS,” Technical Report 91016, Rensselaer Design Research Center, Rensselaer Polytechnic Institute, Troy, New York, 1991.

[Hard93] M. Hardwick, “Implementing Concurrent Engineering Using STEP, EXPRESS, and Delta Files,” *Languages for Manufacturing and Design*, B. Gruver and J. Broudreaux, ed., Springer Verlag, London 1993.

[Hard94] M. Hardwick, “Towards Integrated Product Databases Using Views,” Technical Report 94003, Rensselaer Design Research Center, Rensselaer Polytechnic Institute, Troy, New York, 1994.

[Hard95a] M. Hardwick, B. Downie, M. Kutcher, and D. Spooner, “Concurrent Engineering with Delta Files,” *IEEE Computer Graphics and Applications*, Vol. 15, No. 5, January 1995, pp. 62-68.

[Hard95b] M. Hardwick and D. Loffredo, “Using EXPRESS to Implement Concurrent Engineering Databases,” *Proc. of the Ninth Annual ASME Engineering Database Symposium*, Boston, Mass., September 17-21, 1995.

[Hard95c] M. Hardwick, D. Spooner, T. Rando, and K.C. Morris, “Sharing Manufacturing Information in Virtual Enterprises,” *Communications of the ACM*, February 1996.

[Herb94] A. Herbst, "Archiving of Data in an EXPRESS/SDAI Database," *Proc. of EUG '94 — The Fourth EXPRESS Users Group Conference*, Greenville, South Carolina, October 1994.

[Hsu89] C. Hsu, M. Bouziane, L. Rattner, and L. Yee, "GIRD: A Meta-Database Structure for Heterogeneous Distributed Environments," *Proc. Second International Conference on Computer Integrated Manufacturing*, IEEE Press, New York, 1990.

[Hull87] R. Hull and R. King, "Semantic Database Modeling: Survey, Applications, and Research Issues," *ACM Computing Surveys*, Vol. 19, No. 1, September 1987, pp. 201-260.

[IDEF85] *Integrated Information Support System (IISS), Vol V: Common Data Model Subsystem, Part 4: Information Modeling Manual — IDEF1X.* Report Number AFWAL-TR-86-4006, Volume V, AFWAL/MLTC, Wright-Patterson AFB, Ohio, 1985.

[IGES80] *Initial Graphics Exchange Specification*, Version 1.0, NIST, Gaithersburg, Maryland, January 1980.

[ISO94a] *Industrial Automation Systems and Integration — Product Data Representation and Exchange — Part 1: Overview and Fundamental Principles*, ISO 10303-1:1994 (E), ISO, Geneva, 1994.

[ISO94b] *Industrial Automation Systems and Integration — Product Data Representation and Exchange — Part 11: Description Methods: The EXPRESS Language Reference Manual*, ISO 10303-11:1994 (E), ISO, Geneva, 1994.

[ISO94c] *Industrial Automation Systems and Integration — Product Data Representation and Exchange — Part 21: Implementation Methods: Clear Text Encoding of the Exchange Structure*, ISO 10303-21:1994 (E), ISO, Geneva, 1994.

[ISO94d] *Industrial Automation Systems and Integration — Product Data Representation and Exchange — Part 41: Integrated Generic Resources: Fundamentals of Product Description and Support*, ISO 10303-41:1994 (E), ISO, Geneva, 1994.

[ISO94e] *Industrial Automation Systems and Integration — Product Data Representation and Exchange — Part 42: Integrated Generic Resources: Geometric and Topological Representation*, ISO 10303-42:1994 (E), ISO, Geneva, 1994.

[ISO94f] *Industrial Automation Systems and Integration — Product Data Representation and Exchange — Part 203: Application Protocol: Configuration Controlled Design*, ISO 10303-203:1994 (E), ISO, Geneva, 1994.

[ISO95a] *Industrial Automation Systems and Integration — Product Data Representation and Exchange — Part 22: STEP Data Access Interface*, ISO Document TC184/SC4 WG7 N392, July 1995.

[ISO95b] *Industrial Automation Systems and Integration — Product Data Representation and Exchange — Part 23: C++ Language Binding to the Standard Data Access Interface Specification*, ISO Document TC184/SC4 WG7 N393, July 1995.

[ISO95c] *Industrial Automation Systems and Integration — Product Data Representation and Exchange — Part 24: Standard Data Access Interface — C Language Late Binding*, ISO Document TC184/SC4 WG7 N394, July 1995.

[Koni95] H.P. de Koning, P.P. Almazan, "STEP Application Protocol — Thermal Analysis for Space," *Proc. of the 25th Intl. Conf. on Environmental Systems*, San Diego, Calif., July 10-13, 1995 (also SAE Technical Paper Series #951725).

[Kreb95a] T. Krebs and H. Lührsen, "STEP Databases as Integration Platform for Concurrent Engineering," *Proc. 2nd International Conference on Concurrent Engineering* (McLean,Virginia, August 23-25 1995), Concurrent Technologies Corporation, Johnstown, PA, 1995, pp. 131-142.

[Kreb95b] T. Krebs, Jörg Decker, "Translating EXPRESS Models to the Extended Relational Database Management System POSTGRES," *Proc. of EUG '95 — The Fifth EXPRESS Users Group Conference*, Grenoble, October 21-22, 1995.

[Kros89] U. Kroszynski, B. Palstroem, E. Trostmann, and E. Schlechtendahl, "Geometric Data Transfer Between CAD Systems: Solid Models," *IEEE Computer Graphics and Applications*, September 1989, pp. 57-71.

[Lars89] J. Larson, S. Navathe, and R. Elmasri, "A Theory of Attribute Equivalence in Databases with Application to Schema Integration," *IEEE Transactions on Software Engineering*, April 1989.

[Loff94] D. Loffredo and M. Hardwick, "Efficient Implementation of EXPRESS Information Models," *Proc. of EUG '94 — The Fourth EXPRESS Users Group Conference*, Greenville, South Carolina, October 1994.

[Loff95] D. Loffredo, “Product Data Exchange with STEP,” *E-COMM*, Vol. 1, No. 1, August/September 1995, pp. 71-77.

[Loom87] M. Loomis, *The Database Book*, Macmillan Publishing Company, New York, 1987.

[Loom95] M. Loomis, *Object Databases, The Essentials*, Addison Wesley, 1995.

[Mead89] M. Mead and D. Thomas, “Proposed Mapping from EXPRESS to SQL,” Rutherford Appleton Laboratory, May 1989.

[Mikh97] L. Mikhajlov and E. Sekerinski, “The Fragile Base Class Problem and Its Impact on Component Systems,” *Proc. of the Second International Workshop on Component-Oriented Programming (WCOP '97)* , Jyväskylä, June 9, 1997. (Proc. available at http://www.tucs.abo.fi/publications/general/G5.html)

[Morr90] K.C. Morris, “Translating EXPRESS to SQL: A User's Guide,” Technical Report NISTIR 4341, National Institute of Standards and Technology, Gaithersburg, Maryland, May 1990.

[Mull93] J. Muller and G. Smith, “A Pre-Competitive Project in Intelligent Manufacturing Technology,” *Proc. of AAAI ‘93 Workshop on Intelligent Manufacturing Technology*, 1993

[Nijs82] G. Nijssen and D. Vermeir, “A Procedure to Define the Object Type Structure of a Conceptual Schema,” *Information Systems*, Vol. 7, No. 4, 1982.

[Nijs89] G. Nijssen and T. Halpin, *Conceptual Schema and Relational Database Design: A Fact Oriented Approach*, Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[Obje94] *ObjectStore Reference Manual*, Part No. 300-100-001 3C, Object Design, Inc., Burlington, Mass., 1994.

[ODMG93] Rick Cattell, “Object Database Management Group Project Summary,” Sun Microsystems, Mountain View, Calif., March 1993.

[OMG93] R. Soley, “Object Management Group Project Summary,” Object Management Group, Framingham, Mass., March 1993.

[Open92] *OpenODB Reference Document*, Part No. B2470A-90001, Hewlett Packard, Cupertino, Calif., 1992.

[Owen93] J. Owen, *STEP: An Introduction*, Information Geometers (47 Stockers Avenue, Winchester SO22 5LB, United Kingdom), 1993.

[PDDI84] *Product Data Definition Interface*, Technical Reports DR-84-GM-01 through DR-84-GM-05, CAM-I Inc., Arlington, Texas, 1984.

[PDES91] *A High-level Architecture for Implementing a PDES/STEP Data Sharing Environment*, Technical Report PTI017.03.00, PDES Inc., Charleston, South Carolina, May, 1991.

[PDES97] *Recommended Practices for AP-203*, Internal Document, PDES Inc., Charleston, South Carolina, July 1st, 1997. (Also available from ftp://pdes.scra.org/pub/recprac/)

[POSC92a] *InfoPOSC Newsletter*, Vol. 3, No. 3, Petrotechnical Open Software Corporation, Houston, Texas, 1993.

[POSC92b] C. Allen, “POSC Technical Program Overview,” POSC Document TR-1, Petrotechnical Open Software Corporation, Houston, Texas, March 1992.

[Ragh92] V. Raghavan, *STEP Relational Interface*, Master's Thesis, Rensselaer Polytechnic Institute, Troy, New York, December 1992.

[Rumb91] J. Rumbaugh et al., *Object Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[Sama90] G. Samaras, *A Relational Database as a High-Level Index to a Distributed Object Database in Engineering Design Systems*, PhD Thesis, Rensselaer Polytechnic Institute,Troy, New York, 1990.

[Sand93] D. Sanderson and D. Spooner, “Mapping between EXPRESS and Traditional DBMS Models,” *Proc. of EUG ‘93 — The Third EXPRESS Users Group Conference*, Berlin, October 2-3, 1993.

[Sand95] D. Sanderson, *Loss of Data Semantics in Syntax Directed Translation*, PhD Thesis, Rensselaer Polytechnic Institute,Troy, New York, 1995.

[Saud95] David Sauder and K. C. Morris, “Design of a C++ Software Library for Implementing EXPRESS: The NIST STEP Class Library,” *Proc. of EUG ‘95 — The Fifth EXPRESS Users Group Conference*, Grenoble, October 21-22, 1995.

[Sche94] D. Schenck and P. Wilson, *Information Modeling the EXPRESS Way*, Oxford University Press, New York, 1994. ISBN 0-19-508714-3.

[SET85] *Automatisation industrielle. Representation externe des donnees de definition de produits. Specification du standard d'echange et de transferts (SET)*, Version 85-08, Z68-300, Association Francaise de Normalisation (AFNOR) 85181, Paris, 1985.

[Ship81] D. W. Shipman, "The Functional Data Model and the Data Language DAPLEX," *ACM Transactions on Database Systems*, Vol. 6, No. 1, March 1981, pp. 140-173 (also in *Readings in Database Systems*, M. Stonebreaker, ed. Morgan Kaufman, San Mateo, Calif., 1988, pp.388-404).

[Snyd86] A. Snyder, "Encapsulation and Inheritance in Object-Oriented Programming Languages," *OOPSLA '86 Proceedings*, pp. 38-45, 1986.

[STI92a] *ST-Developer — SDAI Library Reference Manual*, STEP Tools, Inc., Troy, New York, 1992.

[STI92b] *ST-Developer — ROSE Library Reference Manual*, STEP Tools, Inc., Troy, New York, 1992.

[STI92c] *ST-Developer — STEP Utilities Reference Manual*, STEP Tools, Inc., Troy, New York, 1992.

[Totl92] T. Totland, *Translation of Definitions and Data from EXPRESS to C++*, Diploma Thesis, Norwegian Institute of Technology, Trondheim, January 1992.

[Vers93] *Versant ODBMS System Reference Manual*, Versant Object Technology, Menlo Park, Calif., 1993.

[VDAF86] *VDA Flaechenschnittstelle (VDAFS)*, Version 1.0, Deutsches Institute für Normung (DIN) 66301, Beuth Verlag, Berlin, 1986.

[Whit91] C. Whitehead, "STEP Implementation Prototype Package for LEVEL III Database Prototype using OODB Technology, Digital's Prototype Report," Digital Equipment Corporation, Chelmsford, MA, November 21, 1991.

[Wils87] P. R. Wilson, "A Short History of CAD Data Transfer Standards," *IEEE Computer Graphics and Applications*, Vol.7, No. 6, June 1987, pp. 64-67.

[Wils90] P. R. Wilson, “Information Modeling and PDES/STEP,” Technical Report 90017, Rensselaer Design Research Center, Rensselaer Polytechnic Institute, Troy, New York, 1990.

[Wils91] P. Wilson, “Modeling Languages Compared: IDEF1X, EXPRESS, NIAM, OMT, and Shlaer-Mellor,” Technical Report 92001, Rensselaer Design Research Center, Rensselaer Polytechnic Institute, Troy, New York, 1991.

[Wils93] P. Wilson, “A View of STEP,” *Geometric Modeling for Product Realization*, P. Wilson, M. Wozny, and M Pratt, ed. Elsevier Science Publishers B.V., North Holland, 1993, pp. 267-296. ISBN 0-444-81662-3.

[Winn88] R. Winner, “The Role of Concurrent Engineering in Weapon Systems Acquisition,” Report R-338, Institute for Defense Analyses, Alexandria, Virginia, 1988.

[Zani83] C. Zaniolo, “The Database Language GEM,” *Proc. 1983 ACM SIGMOD Conference on Management of Data*, San Jose, Calif., May 1983.