Using the STEP Standard and Databases in Science

Udo Nink
University of Kaiserslautern
P.O. Box 3049, 67653 Kaiserslautern, Germany
e-mail: nink@informatik.uni-kl.de

Abstract

Database management systems (DBMS) have to provide certain facilities meeting the requirements of scientific and statistical database management. Reviewing problems and promises for current database technology, the STEP standard is assessed for selected aspects of data management, data access, data exchange, and data modeling. STEP-based solutions are proposed for concrete examples of SSDBM especially in the context of the scientific data exchange standard FITS. We introduce and discuss EXPRESS, the modeling language of STEP, and SDAI, the corresponding data access interface. The performance of navigational access provided by SDAI is considered a crucial aspect. Exploiting the code generation mechanism used to instantiate SDAI for a given programming language - we call it a generated call interface - an adequate software architecture on top of an ODBMS as well as STEP-specific optimizations are proposed.

1. Introduction

Researchers of different fields have identified a large set of general and domain-specific characteristics with scientific and statistical database management (SSDBM) in the past [4, 8, 15, 18, 24, 25, 26, 28, 30]. From [26] we learn that an integrated working environment (IWE) must be able to **manage** distributed and heterogeneous data sources (short sources) like files and databases as well as tools like functions of a statistical package in a uniform way. It must also allow to efficiently **access** (find, read, update, associate) them. Moreover, centralized data management helps to avoid redundancy and alleviates reuse of shared data. Reuse may be further supported through explicit data **exchange** between such environments introducing visible redundancy. Thus, we identify three very important pillars of an IWE (see figure 1).

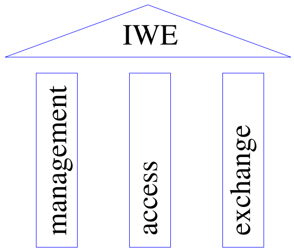

Figure 1: IWE pillars

- *data management:* Space- and time-efficient **storage mechanisms** and **indexing** [4, 26, 28] (often spatial at various levels of grid) are needed to cope with searching within the giant data flood produced by, for example, radio astronomy. Moreover, the data pool may get too large for secondary storage devices (harddisks) or users just might wish to hold backups on tapes. Therefore, tertiary storage management [18, 28] and especially **archiving** become important.
- *data access:* Data access varies from simple, explicit **navigation** between single objects to powerful **queries** needing to describe what to do with which data without giving a recipe how to do it. Different **access patterns** like ordered vs unordered and unrestricted vs restricted access occur. Moreover, a two-step processing [4, 30] seems to be very common: first, a description of some astronomical image or the book-keeping data of some experiment is searched; second, part of the image or some experiment measurement is accessed through an index. Various **access granules** like attributes, simple objects, complex objects, and collections are often found. A complex object may contain a net of simple objects and a collection may represent a multidimensional array as a whole, subset or slice. Data from current as well as legacy applications must be accessible from several programming languages.
- *data exchange:* Tools demand for data exchange of sources without information loss. These **heterogenous**

Published in Proc. 9th International Conference on Scientific and Statistical Database Management, August 1997, Olympia, Washington.

sources may be databases of different DBMSs or ordinary files of different formats. In addition, **interpretation** of such data may vary.

Data exchange often presumes files. Files represent a future legacy problem, since distributed objects are underway with an incredibly fast pace. Consequently, data exchange or distribution (often in the sense of broadcasting) soon will not have to undergo the following steps anymore: copying the data into files, storing the files on a diskette or zip medium, physical transportation to another place, and, finally, copying into another computer and another IWE. You will not even need "ftp" when having replaced those hard transport media by the rather soft internet then. Instead, applications themselves will be able to transparently access distributed objects online. Huge quantities of data like image databases will be logically stored only once somewhere in the Internet and clients will ask for the portions they currently need. Nevertheless, file-based data exchange will not vanish at once, since paradigm shifts have always been slow. So far, existing file-based scientific data exchange standards (FITS, HDF, NetCDF and others [37]) have concentrated on some data interchange format (short **DIF** [18]). Only a few worry about data access and data management, though providing rather limited solutions.

On the other hand, database technology is very strong in centralized data modeling and schema management, delivers one global view to data, and, therefore, alleviates overview and exchange of data as well as book-keeping and implementation of applications. Moreover, sharing of smaller-than-file data granules (a row in a table for instance), concurrency of many applications, recovery from system failures, and querying very large data sets are supported. Query optimizers achieve performance by taking into account different storage strategies, access paths and processing algorithms when constructing alternative execution plans for a query to be evaluated. But, unfortunately, database technology did not care about DIFs.

Thus, these worlds still have to be merged. There are at least two ways to accomplish this: completing DIFs with database technology or vice versa.

Completing DIFs with current database technology would be quite a job to do. In the recent past, object database management systems (ODBMSs) have always been trying to catch up with the relational database world. 1991, the year ODMG (Object Data Management Group [2, 3]) has been conceived by a group of frustrated ODBMS vendors, relational DBMSs were acknowledged to have a ten year advantage over ODBMSs. Things have changed in the meantime and object-orientation (short OO) has run into our lives, but ODBMs surely did not catch up. They rather seem to get swallowed by relational vendors that diligently define quite different products extended by these or those OO features. All products are, amusingly, called universal server or alike. Obviously, this approach is slow. But scientific and other data exchange standards very often follow this path, anyway, trying to invent the wheel once again.

Completing database technology with DIFs seems easier at a first glance. But there are at least two problems. First, database vendors cannot be influenced much, except by participation in standardization efforts. Second, there are lots of DIFs to implement.

Therefore, we have to search for another solution allowing to shift much of the implementation effort to existing database technology. This is possible by hiding underlying storage from application development. Application developers usually do not want to see underlying storage. They do not even want to see transactions or alike, which is even harder to realize and out of the scope of this paper. To hide data storage a stable interface is needed allowing to replace or even switch between concrete implementations. Ideally, freely switching between relational and OO systems and vice versa, or the concurrent use of both worlds is provided.

Now this is the way of STEP (ISO 10303, Industrial Automation Systems and Integration - Product Data Representation and Exchange [23, 34]). STEP has emerged from long experience with electronic data interchange. Though initially concentrating on product data, its basic and most important concepts are generic and independent of application domain. Consequently, it has soon become a framework to describe and specify data. For this purpose a powerful data modeling language named EXPRESS [11] has been developed which is ISO international standard since January 1994. This quite unique feature in the set of data exchange standards represents a formalism to describe data and is the very basis of the whole standard. Most important parts of the standard define, always using EXPRESS, the STEP Data Access Interface (SDAI [13]), the clear text encoding [12], and lots of schemas supporting data design. It is these mentioned parts that support data management, data access, and data exchange independent of database technology. Therefore, let us set the pillars illustrated in figure 1 on top of a solid foundation (see figure 2):

- *data modeling:* Scientific and statistical applications make heavy use of complex objects and operations. Data may be unstructured, time-ordered, derived, or multimedia and may have complex relationships. Thus, powerful definition of **data types** and **relationships** is needed. As observed before, there are two different notions of data semantics: **meta** data (annotation, description, book-keeping) vs. **real** data (analysis, measure) [4]. Corresponding schemas contain very complex as well as simple parts, and corresponding data sets are large or very

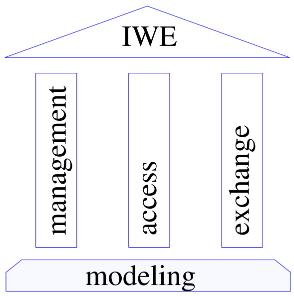

Figure 2: IWE pillars (2)

large, respectively. Additionally, data from different sources are often modeled in different ways (**heterogeneous schemas**) and porting of data and code is a pain.

The most important pillar of the three on top of the foundation is the pillar of data access. Access through SDAI means access to data which has been defined in EXPRESS, no matter where the data resides. Data exchange may thus be realized using one generic SDAI program that can copy data from one repository (a file or database including real data and STEP meta data) to another. Data management is hidden by SDAI and has lead to many STEP implementations some of which have been reported in [8, 10, 16, 17, 20, 32]. They show that implementation effort may be balanced between database technology and the STEP-layer itself. Archiving has been discussed in [8]. Distributed data and federated repositories are studied in [10]; there, an architecture is proposed that allows for SDAI-based access to different DBMSs; a schema mapping component and a mapping description language are central for this approach; the system is operational in a prototypical variant-bill-of-materials application of a leading german automobile vendor. For investigations of implementations on relational, extended-relational, and object-oriented DBMSs see [16, 17]. Extending EXPRESS to become a fully object-oriented programming language is discussed in [20]. EXPRESS and SDAI are enhanced with semantic relationships in [32].

In contrast to these systems we concentrate on a design optimally supporting application optimization. The advantages of navigational access for certain access characteristics have early been proved in the OO1 Benchmark [7]. On the other hand, unthinkingly used navigational access may lead to horrible response times. And as we have shown in [21], additional performance losses are to be expected for the extra SDAI software layer if not serious precautions are undertaken. We, therefore, exploit abstractions in STEP that allow us to early bind information in order to improve performance (and detect errors as soon as possible). Our second goal is to allow for replacement of underlying database technology. Thus, the resulting software architecture of our prototype minimizes overall system dependency. Since portability contradicts performance, we have made a compromise by implementing only few parts that are most important for efficiency in a system dependent manner. Thus, integrating our thoughts into what we define “generated call interface”, allows us to stay with a layered software architecture <u>and</u> be performant.

In the following section we will introduce the STEP standard. We then concentrate in section 3 on applying STEP to SSDBM and, especially, to the scientific data exchange standard FITS (Flexible Image Transport System). There, we discuss the two most important parts of the standard from our point of view: EXPRESS and SDAI. Afterwards, we concentrate on optimization of SDAI-like access and propose an adequate architecture as well as STEP-specific optimizations before we conclude our work.

2. STEP

ISO 10303 - STEP [23, 34] is developed by ISO TC184/SC4 Industrial Automation Systems and Integration. Its objective is to neutrally and system-independently describe products throughout their life cycle from design to maintenance. However, the resulting formalisms have made it superior to simply file-based data exchange formats; it has become the transparent basis for implementing and sharing product databases. As [8, 27, 31] have shown and will be shown here, there are many domain-independent aspects applicable to SSDBM and other non-product application domains as well. STEP is organized in a series of parts which may be grouped into five main groupings[1].

- Description methods (see figure 3 [35]) form the very basis. Part 1, Overview, contains universal definitions, and part 11, EXPRESS Language Reference Manual, describes the data-modeling language and, thus, the abstract data model.
- Implementation methods describe the mapping from formal specifications to a representation usable for implementation. Part 21 represents the file-based clear text encoding (which is what DIFs traditionally have aimed at), parts 22 - 26 describe the language-independent part of SDAI plus some language bindings (C++, C, FORTRAN, IDL). A mapping to Java is also under development.
- Conformance methodology provides information for testing of software-conformance to the standard and, especially, how to create abstract test suites and which methods be tested.

1. Thanks to Jim Nell allowing us to reuse his “STEP on a Page”.

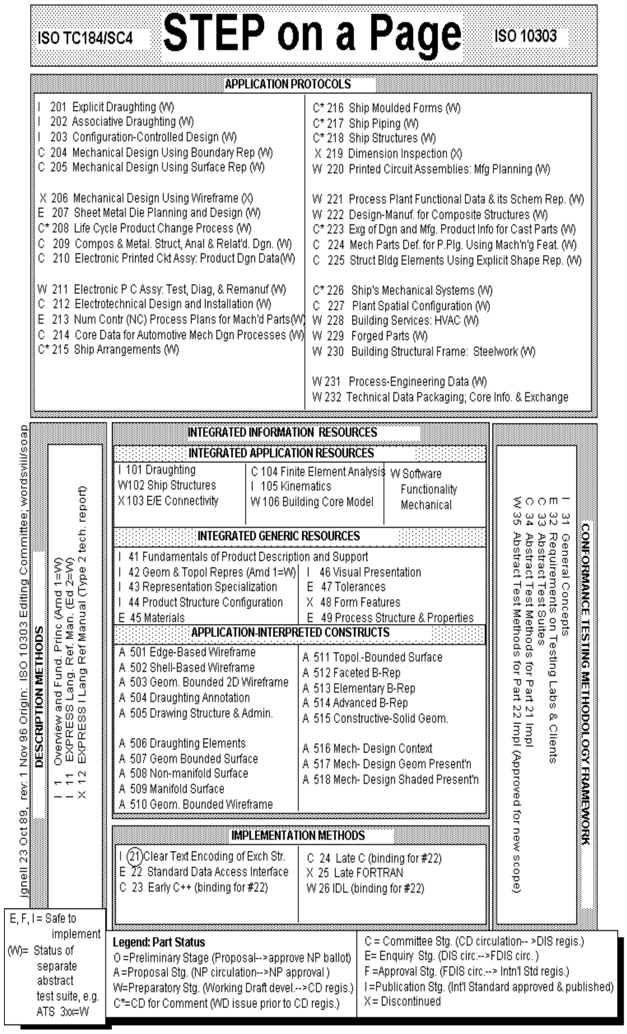

Figure 3: STEP on a page

- Integrated information resources comprise concrete data models divided into several layers to serve as basis for the definition of application protocols (see below). The lowest layer (regarding the part numbers), integrated generic resources, contains overall used generic entities. The integrated application resources have slightly more context. They alleviate application protocol integration and enable interoperability. Application-interpreted constructs are even more precise and may be used to define identical concepts across application protocols.
- Application protocols, finally, describe specific product-data applications, that is, STEP applications will normally be implemented on top of an application protocol reusing there defined types.

3. Applying STEP to SSDBM

There exist numerous scientific data formats. [37] gives an overview over about 20. [36] is a very good starting point for FITS (Flexible Image Transport System) that will be compared to STEP in the following discussion. FITS is a worldwide accepted file-based data exchange standard in the astronomy community that helps crossing installations whose internal formats and hardware differ. The International Astronomical Union FITS Working Group (IAU FWG) was given authority over FITS matters by the 1988 IAU Assembly. FITS has evolved from a graphic format in the very beginning to an abstract data format. Though it does not provide for ready-to-use code, several general software packages are available. In the following we will introduce EXPRESS and SDAI and apply them to FITS.

3.1. Data modeling - EXPRESSing data

We view EXPRESS [11] as DDL (data definition language of a DBMS). Objects are defined through strongly typed *entities* aggregating properties (*attributes* and *constraints*). Attributes may be constants, be of data types like integer, collections (also called aggregates), named types, enumerations, selection types similar to unions in C, or generalized types. Instances of entities are unique through a non-visible, system-defined identifier. In addition, optional keys may be defined by the user. User-defined *relationships* are explicitly modeled by entity-valued attributes; for such a given attribute an inverse attribute may be defined at the partner entity to further restrict the existing relationship and to allow “backward” read access.

Besides these features that are also present in most other object models, EXPRESS provides multiple inheritance; all properties of several parent entities are inherited to a child entity, and each single property may be refined in the child. Another, quite unique feature, is multi-class membership (also called *complex entity type*), an elegant solution to implicitly define new combinations of existing classes using predicates; thus, many explicit definitions exploiting multiple inheritance may be avoided; as example, think of persons that, naturally, have more than one role at a time and migrate between roles during their lives. To represent all possible combinations of scholar, worker, football-player, and basketball-player in C++ you would have to define 10 extra classes; see also below for an application to FITS.

EXPRESS is ‘only’ structurally object-oriented, that is, object behaviour like methods in a class are not available. However, *functions* and *procedures*, today only used for implementing derived attributes and integrity constraints (also called rules), may be defined using common procedural language statements and built-in functions. Except for the possibility to write an executable main program, EXPRESS may be viewed as a programming language.

A *schema* may contain definitions of value types, entities and integrity constraints. Several schemas may exist for a repository, and they may reuse each other on type granules. Schemas are also used as scope for the evaluation of integrity constraints, functions and procedures.

Note that support for metadata exists, but in a slightly different sense than in SSDBM: special entities exist describing the structure of any user-defined entity to allow for schema-independent access. Metadata in the SSDBM sense carries application semantics. Therefore, it has to be modeled in EXPRESS, too. Thus, STEP delivers a kernel framework to build an extended SSDBMS. And by definition of application protocols domain scientists may participate in the standardization process, thereby integrating domain-specific data structures and semantics.

3.2. Modeling the example

The presented schemas in figures 4 and 5 are redesigned from the Astronomical Image Processing System (AIPS) of Allen Farris [33]. The underlying object model is the one of C++. Since most ODBMSs are based on C++, our redesign in EXPRESS will help to easily see the differences. A FITS file is composed of a sequence of Header Data Units (HDUs). The header of an HDU consists of a list of "keyword=value" statements describing the format of contained data. Additionally, information like instrument status or history may be included here. The data section of the HDU may contain a digital image, a table, or a multidimensional matrix depending on the chosen subtype of HDU. Many constraints how the data has to be organized and formatted exist in the standard; we will only refer to those that we need here. To keep things simple, we also restrict ourselves to the shaded region in figure 4.

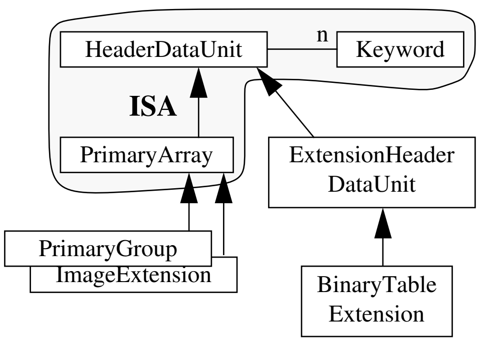

Figure 4: FITS header data units

The class hierarchy in figure 5 encapsulates access to FITS files. We have illustrated our version only to show the benefits of multi-class membership. If using the following clause, we may omit the explicit definition of the classes below the horizontal line in figure 5:

ENTITY Media **SUPERTYPE** OF
(**ONEOF**(BlockInput, BlockOutput)
ANDOR
ONEOF(Disk, Tape9, Std));

Note that "ONEOF" is a class selector and works like "xor" and "ANDOR" is the class constructor. "A ANDOR B" results in classes A, B, and AB.

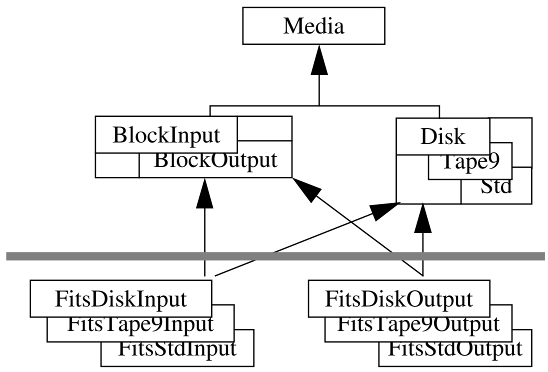

Figure 5: FITS I/O

Let us now concentrate on HDUs again. Figure 6 illustrates a possible design for some FITS structures in EXPRESS. Some useful aspects shall be pointed out.

First, management of identifiers that may participate in relationships is automatically done by the system. Only "Keyword" shall have "name" as user-defined identifier (or key), in order to support fast searching from outside. We accomplish this by "UNIQUE", a local rule. Together with the automatic extent-management for entities the set of allowed keywords may thus be controlled.

Second, the relationship between "HeaderDataUnit" and "Keyword" is represented by an attribute named "keywords" referencing a list of "Keyword". To support analysis of occurrences of keywords in other HDUs, an additional "INVERSE" path may be added to avoid searching or joining known from relational systems (see set-valued attribute "used_in" in "Keyword").

Third, "PrimaryArray" contains a derived attribute computed by the function "getRawData" that will be discussed later. Since this function also expects parameters from outside, we add the list-valued attribute "tmpidx" that may hold up to 999 values (a FITS constraint) depending on the number of existing dimensions of the "data" array. The

```
SCHEMA fits;
REFERENCE FROM basics (SomeValue);
TYPE HDUType = ENUMERATION OF
  (NotAHDU, PrimaryArrayHDU, UnknownExtensionHDU);
END_TYPE;
ENTITY HeaderDataUnit;
  id: INTEGER; nodims: INTEGER; size: INTEGER;
  hdutype: HDUType;
  keywords: LIST [0:?] OF Keyword;
END_ENTITY;
ENTITY PrimaryArray SUBTYPE OF (HeaderDataUnit);
  bscale: REAL; bzero: REAL;
  data: LIST [0:?] OF SomeValue;
  factor: LIST [0:999] OF INTEGER;
  tmpidx: LIST [1:999] OF INTEGER;
  DERIVE pdata: REAL :=
       bscale * getRawData(data, factor, tmpidx) + bzero;
END_ENTITY;
ENTITY Keyword;
  name: STRING; comment: STRING;
  pdata: SomeValue;
  INVERSE
    used_in: SET [0:?] OF HeaderDataUnit FOR keywords;
  UNIQUE UR1: name;
  WHERE ok: check(name) = TRUE
END_ENTITY;
END_SCHEMA;
```

Figure 6: EXPRESS schema for FITS

function “getRawData” searches a value from the array and returns it multiplied with some “bscale” and added to some “bzero” in order to convert array values into true physical values in case of different formats.

Fourth, we do not need any extra data in order to make file IO work, because this is clearly the task of a converter between the STEP clear text encoding (part 21 [12]) and FITS. To write STEP data out of some repository into a file according to part 21 (which is just another repository), a generic copy-routine can be written or used (we wrote one with an older version of SDAI). This routine may be developed to be independent of any given schema, because of three prerequisites: EXPRESS defines the abstract data model common to all repositories, one schema may define a concrete data model for several (here both) repositories, and SDAI allows meta-data-based access to both (see next subsection).

Fifth, there are many, often low-level, constraints on FITS data or corresponding format in a file. We do not have to adopt those that are only relevant to the pure file format. The restriction that there must be 36 card images of fixed length (unused space filled up with blanks) per header record of an HDU is a good example. Other constraints, nevertheless, are worth a thought. The keyword format, for instance, allows only digits, upper case Latin alphabetic characters, the underscore, and the hyphen. Then there are mandatory keywords that must appear in a header. For both problems we can define rules; a local rule to check a keyword (see WHERE-clause “ok” in “Keyword” calling some “check” function) and a global rule to check the “keywords” of an HDU. Since global rules are quite similar to functions (plus optional WHERE-clause) we skip an example.

Sixth, we omitted the value indicator of a keyword. Normally, if it contains “= “, an associated value field must exist (here “pdata”). Since we have predefined null values in EXPRESS, we do not need this field and, instead, may test the value of “pdata” directly. We use the predefined NVL-function (FUNCTION NVL(VALUE:GENERIC:T; SUBSTITUTE:GENERIC:T):GENERIC:T;) that delivers the “VALUE”, if not indeterminate, and the “SUBSTITUTE” otherwise. Regard that “NVL” uses “GENERIC” parameters accepting any type. The type-label “T” is used to enforce that both, parameters and the return type, share exactly the same type.

EXPRESS has lots of predefined functions, but we have to break the discussion and move on to SDAI.

3.3. Data access - SDAI

In analogy to EXPRESS, we view SDAI as DML (data manipulation language). It provides its functionality through an API (application programming interface) like most ODBMSs do. Moreover, functionality is roughly the same.

SDAI defines abstractions like *model* (a named physical object container) or *schema instance* (logical collection of shared models obeying to the same schema). *Repositories* (databases) may contain several schema instances. Five predefined schemas provide for metadata corresponding to user-defined classes (dictionary schema), session control (session schema), application data management (population schema), predefined classes (data type schema), and parameters (parameter data schema).

Principally, the programming model may be characterized by root access to objects or data containers, navigation from object to object, and, supporting different access patterns, rudimentary queries and iteration on different collection types (set, bag, list, and array). Data access must occur inside transaction boundaries where SDAI only enforces atomicity and durability (“A” and “D” of the ACID paradigm, [9]). Atomicity means that either all calculations of a transaction are guaranteed to come through or none. Durability means that results survive process boundaries and do not get lost in case of a system crash. Rules centrally defined in EXPRESS may only be validated explicitly; consequently, the "C" in ACID, consistency, is not automatically ensured by the system. This is quite dangerous because integrity checking, thus, is possibly subject to many decentral applications. Moreover, STEP does not explicitly talk about the "I", isolation, that is, it does not force simulation of single-user sessions in a multi-user environment. Therefore, applications may still behave quite differently on different STEP-systems.

The mentioned two-step processing in astronomy and other scientific as well as statistical domains may be well served by storing catalog data and raw data in different schema instances and models. To allow references between instances of different models (pointer from catalog entry into image base for example) the schema instances must overlap in the corresponding portions of their schemas.

Different access granules range from simple-typed attributes over set-valued attributes (using iterators), objects, and object sets (especially entity extents inside models) to entire models and schema instances (used as scope for validating rules).

Hiding from existing storage mechanisms of main memory, file or database system SDAI defines data structures and operations supporting access to heterogenous STEP repositories (data in main memory, files, or databases). These interface specifications are also independent of programming languages. In addition, language-specific bindings exist mapping the specified interface to existing programming language constructs, thereby exploiting the capabilities of the current language. In consequence, the language bindings still have much in common and the programming model is the same regarding the SDAI part.
Unfortunately, the work on the FORTRAN binding has been cancelled in October 1995 due to lack of activity! There are, principally, two possible actions now: revive the binding by a proposal or, because a binding to OMG's IDL is currently under work (and will not be cancelled), force a FORTRAN binding to IDL.
The existing binding to C++ is twofold and provides an early binding for schema-dependent access as well as a late binding for schema-independent access (useful for browsers or converters between file-formats).

Last but not least, the main part of SDAI and some bindings will soon be international standard, and, therefore, portability of applications and uniform data access will be guaranteed, to a high degree, across STEP-DBMSs.

3.4. Programming the example

We have introduced the function "getRawData" in the FITS example. Figure 7 illustrates the corresponding definition in EXPRESS. The function accepts a list of values (the "data" array of an HDU), a list "idx" of coordinates, and a list "factor" of dimension sizes. The local variable "i" is needed for the repeat loop where the "offset" into the linear array is calculated. Then, the array is accessed and the value returned.

```
FUNCTION getRawData(data: LIST:x OF SomeValue;
     idx: LIST OF INTEGER;
     factor: LIST OF INTEGER)
     : REAL;
LOCAL i: INTEGER; offset: INTEGER;
END_LOCAL;
  offset := idx[LOINDEX(idx)];
  REPEAT i := LOINDEX(idx) + 1 TO HIINDEX(idx);
     offset := offset + idx[i] * factor[i];
  END_REPEAT;
  return(data[offset]);
END_FUNCTION;
```

Figure 7: Access to FITS data

Now, the reader may ask, what is the role of SDAI? The answer is rather simple: you either program the functionality centrally with EXPRESS in a schema <u>or</u> with SDAI for a given progamming language. The former solution may be called from SDAI which mainly provides an interface to EXPRESSed entities and their attributes including derived attributes and, therefore, procedural elements. This approach is to be preferred for multi-language environments to centralize application semantics and, thus, to allow reuse by automatically generated code. The latter approach, coding in SDAI, may, on the other hand, lead to better exploitation of existing software packages.

```
Real getRawData(Somevalue__list_ptr data;
    INTEGER__list_ptr idx, factor) {
  int i = idx->Lower(); int offset = idx->GetByIndex(i);
  for (i = i+1; i <= idx->Upper(); i++)
    offset += idx->GetByIndex(i) * factor->GetByIndex(i);
  return data->GetByIndex(offset);
}
void sameKeywords(Headerdataunit_ptr hdu, hdu2;
    Headerdataunit__set_var& r) {
  hdu->keywords->query("ENTITY IN used_in", hdu2, &r);
  Headerdataunit__iterator it(r);
  for (it.Beginning(); it.Next(); )
    doIt(it->GetCurrentMember);
}
```

Figure 8: Sample SDAI code

Figure 8 contains a solution for "getRawData" using SDAI from within C++. Its structure is very similar to the solution in figure 7. In addition, we implement a function "sameKeywords" to illustrate the use of SDAI queries and corresponding iterators. The function matches keywords of two given HDUs using an SDAI query (the query string is evaluated against each entry in the aggregate; "ENTITY" is placeholder for the parameter "hdu2" to the query). An iterator on the result of the query containing keywords common to both HDUs is used to call "doIt" (not shown) for each of these keywords. Note that all names in the sample code that also occur in the schema result from code generation; all underlying types and methods are complete and ready to use.

4. Tuning call interfaces by code generation

In [22] we have argued for call interfaces being the best choice to serve application programming altogether. Coding with call interfaces (IMS, ADABAS, or ODBC [19]) follows the syntax of the chosen programming language, and applications use database functionality through (often generic) routines of given libraries. Thus, no preprocessing concept or host language compiler extension is needed as in the case of "embedding" (eSQL, [19]) or "integrated language" (persistent ALGOL, [1]). Therefore, standard compilers suffice and compilation speed is optimal. Further, internal coding of the interface is easy due to the decoupled approach. In general, the learning overhead is rather low and depends more on the complexity of the database data model. Moreover, call interfaces have only recently been subject to standardization as for ODBC [5], ODMG [2, 3], SDAI (STEP Data Access Interface [13, 14]), and - last but not least - the future call level interface of SQL [29].

On the other hand, coding of applications with call interfaces has usually been low-level and cryptic. Moreover, access control and error detection have been poor, and, especially, performance has been poor because of indirections.

In the following, we will concentrate on improving the most important aspect: performance. In [21] we have shown that SDAI introduces additional performance penalties when straightforwardly implemented on top of ODBMSs. This is because its functionality is very similiar to that of APIs of ODBMSs and it does not introduce more powerful concepts that could be exploited. But it is, nevertheless, possible to improve performance and even to outperform the underlying ODBMS in some cases. We bind types and operations or both combined as abstract data type (ADT, class) early, that is not later than application compile time for a programming language with static type checking. To prevent precompilers or compiler extensions, we apply code generation to complete the predefined interface, which is described through rules, and its implementation.

In our prototype code generation may be influenced by additional integration descriptions. These are independent of the underlying ODBMS or host programming language, in order to allow for easy porting. SDAI, ODMG, and the VStore-API (our prototype of an API for a DBMS tailored to adequate management of explicit complex-object versions, [22]) exploit different instantiations of this integration technique which we call *generated call interface*. Part of these APIs may (optionally) be generated out of schemas (for all approaches), out of integration description (in our SDAI prototype), and queries (in the case of VStore).

4.1. Architecture

Our first iteration of a STEP-DBMS layer resulted in a fully functional C++ early binding on top of Ontos. Comic [10] has been ported to our implementation. The prototype is still under development regarding performance tuning.

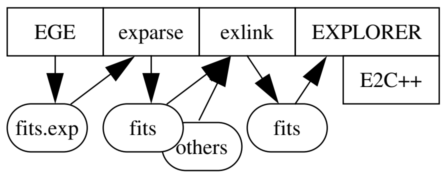

Figure 9: Schema development

First, we introduce the development environment containing our layer as a central part. Figures 9, 10, and 11 show the software components and the data (or file) flow between them from schema development to application linking. Figure 9 comprises the steps to prepare code generation using a commercial EXPRESS toolkit [6]. A user develops a schema with *EGE* (EXPRESS-G graphical editor), parses the resulting file with *exparse* producing an internal representation which is completed with *exlink* evaluating external schemas. The result is accessible through the *EXPLORER* library on top of which we have built our E2C++ code generator. This approach provides flexibility to react to changes to SDAI, which have been frequent in the past, and to the differences between the target ODBMSs.

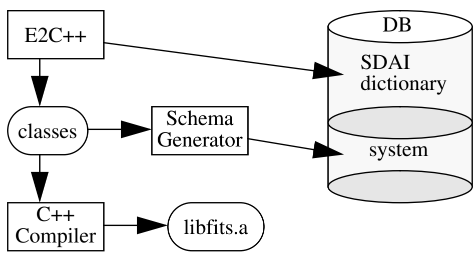

Figure 10: Code generation

Figure 10 contains the steps to be performed until application compilation. E2C++ generates C++ classes (one header and one implementation file for each class) for the specified ODBMS basically following the mapping from EXPRESS to IDL and farther to C++. The schema generator of the ODBMS extracts corresponding meta data usually from the header files and stores it in a database. Additionally, E2C++ initializes the SDAI (data) dictionary. The C++ classes may now be compiled by a standard C++ compiler and archived in a library (libfits.a).

Figure 11 illustrates the remaining steps. The application code is compiled with a standard C++ compiler and linked with the generic and the schema-specific SDAI libraries to form an executable program.

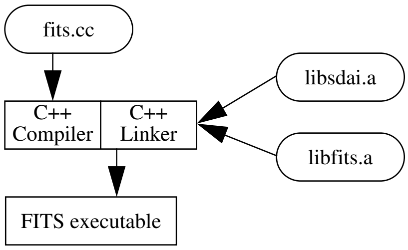

Figure 11: Application building

Our main design decisions were based on the following observation. The frequency of calls to SDAI operations generally decreases with increasing complexity. The simplest and most frequent operations are construction, destruction, and assignment of simple-type attributes. Still simple and quite frequent operations are attribute access and test methods. More complex and less frequent operations are construction and destruction of entity instances. Most other operations are even more complex and less frequent. Since the entire SDAI environment is specified in EXPRESS, too, we coded the latter operations in SDAI in order to keep them portable.

The resulting modules depicted in figure 12 have different degrees of dependency to the DBMS. When changing the DBMS the simple types including object references have to be completely reimplemented. And the code generator has to be adapted to produce correct class implementations that depend on the simple types. On the other hand, code generation allows for completely regenerating schema-specific classes (like "Keyword") and DBS-independent classes (like "Model"), and it minimizes reimplementation for DBS-dependent classes (like "Transaction").

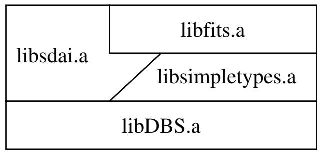

Figure 12: Software layers

4.2. Performance

Our intention is to adequately provide high performance through alternative *storage* and *caching with prefetching*. Currently, basic mechanisms to control code generation have been provided and the code that has to be replaced has been identified. We are lacking a complete language to describe the integration and, of course, a complete implementation right now.

Wherever data containers are defined as ADTs (like a model or some set), implementations underneath may be replaced as long as they obey the interface. The same holds for single objects and attributes, since access to these must occur through methods. Therefore, attributes may be *embedded* as part of the physical memory layout of the object or *referenced* with a reference object in-place. Additionally, attributes may be *materialized* or *calculated*. Embedding of attributes needs less space and decreases communication, because such an attribute is fetched together with its object. It is usually used for rather small, private data (not existing without the surrounding object) of fixed or bounded size. On the other hand, referencing allows for finer grained locking and for delay of fetching "expensive" attributes (big images or high conversion overhead in federated systems). Usually, unbounded dynamic data (some set) and shared data like entity instances are stored by reference. However, it may be useful to skip these heuristics. Private data may be stored by reference to avoid redundancy. Or, the other way round, instances may be embedded knowing that they actually will not be shared or, in case of data distribution, introducing redundancy for the sake of faster access. The decision to materialize or calculate an attribute is especially useful for inverse attributes of explicit reference attributes. The collection of inverse references is either explicitly stored or calculated through some search routine. Read-only access to both, the reference attribute and the corresponding inverse attribute, vote for materialization. On the other hand, frequent updates to the reference attribute may result in expensive automatic updates of the inverse attribute; in this case, calculation is better. Another example for materialization are entity extents inside models. An entity extent of a model logically contains all instances of a specific type and <u>all</u> of its subtypes that are stored in the model. Therefore, we may choose between redundant or disjoint storage for the extents corresponding to the nodes of the inheritance hierarchy of the entity.

Caching is used to speed up consecutive access to previously accessed data. Instances and referenced values may be cached at the client. Computed values like derived or inverse attributes may be cached after the first computation. This requires automatic recomputation, if the value has changed in the meantime, using, for example, some invalidation mechanism triggered by interfering changes of the same user (remember that other users are isolated in our case). Since we allow for the choice of storing such results locally to an instance or in a global table, function result caching as described in [28] is directly supported. Caching may be controlled by a prefetching mechanism discussed in the next paragraph.

```
<integration> ::=  [ <entity> | <global> ] ‘;’
     <entity> ::=  ENTITY <name> ‘;’
                     { <entity_rule> ‘;’ }
                     { <attr> ‘;’}
                   END_ENTITY
<entity_rule> ::=  ON <entity_op> DO <action>
       <attr> ::=  ATTRIBUTE <name> [CALC | REF | EMBED] ‘;’
                     { <attr_rule> ‘;’ }
                   END_ATTRIBUTE
  <attr_rule> ::=  ON [ <attr_op> ] DO <action>
  <entity_op> ::=  VALIDATE
    <attr_op> ::=  [ READ | WRITE | TEST | UNSET |
                     ITERATE | QUERY [<string>] ]
     <action> ::=  [ ACTIVATE | DEACTIVATE |
                     WRITEBACK | NOP ]
                     [ ATTRIBUTE <name> [ ‘,’ <integer> ] |
                        SELF | DEEP <integer> |
                        QUERYRESULT | QUERY <string> ]
     <global> ::=  ON <op> FOR <name> DO <action>
         <op> ::=  [ OpenModel | GetEntityExtent |
                      CloseModel | ValidateSchemaInstance ]
```

Figure 13: Tuning description language

Prefetching optimizes physical I/O and the replacement of cache contents in certain processing situations. Consecutive access may be sped up by decreasing communication with the server (one big fetch instead of many small fetches), thereby caching data in advance that will hopefully be accessed. On the other hand, prefetching may slow down other applications due to reduced transaction parallelism (a conflict may occur between a read and a write transaction) because of locking prefetched data. Therefore, it is very important to specify the working context as precise as possible while keeping overhead as low as possible. Furthermore, we extend control beyond pure prefetching to capture application knowledge when data is no longer needed and may be deactivated. Thus, we identify adequate *events* that trigger *actions* on *data sets*. Events are given by SDAI operations. Actions are ACTIVATE, DEACTIVATE, and WRITEBACK (see figure 13); these operations are found in any client/server system. Data sets may be single objects (SELF for the current instance, ATTRIBUTE for some entity-valued attribute), simple or aggregate attributes (ATTRIBUTE again), logical clusters (DEEP) or query results (QUERYRESULT, QUERY). Clusters may be physical (stored in the neighbourhood on disk) or logical (related through references). Physical clusters may be created using a “magnetic” object. Newly created instances may be stored in its neighbourhood. Unfortunately, SDAI does not support facultative placement operators in create operations; only a target model may be specified. Therefore, we use physical clustering only internally. Logical clusters are specified for prefetching by restricting the depth of navigation (“DEEP 1” would prefetch directly referenced partners only) for all attributes. Further extensions may capture paths through reference attributes.

4.3. Queries and indexing

Two important techniques in DBMSs coping with performance are indices and query shipping. Indices provide for alternative paths for data access allowing a query optimizer to plan data access according to estimated costs. In many ODBMSs only simple indices are supported and evaluation must often take place at the client (data shipping). This is quite costly for calculating a single value over a very large data set. Alternatively, query shipping allows for execution at the server and, therefore, decreases at least network traffic.

Optimizing SDAI queries is, unfortunately, restricted due to their simple logical expressions. Clauses may not be connected for joining, and path expressions must not have aggregate-valued nodes (except for the leaves) yet. Though, indices may be invisibly used to speed up implicit content-based access in SDAI as in operations like “create instance in model <aname>” that imply searching in dictionaries.

In additon to function result caching, Segev and Chatterjee [28] support query optimization through function indexing (storing known function results). Thus, the optimizer may choose between computing or index lookup for a set of function calls. For SDAI queries we must program a function to first look up its result in the index and, if successful, omit the computation. But, in our prototype, we are able to avoid this explicit programming that has to be done by the application programmer, because of the intermediate code generation step. A description outside SDAI, whether an index should be used or not for instance, then leads to transparent generation of extra code into the function bodies.

5. Conclusions

We have assessed STEP-DBMSs for SSDBM. Problems and requirements have been discussed for data management, data exchange, and, in detail, for data modeling and data access. It has been shown that STEP, though designed for product data management, provides many concepts that make it, first, applicable to SSDBM and other domains and, second, superior to existing scientific data exchange standards. Among these concepts are: the data modeling language EXPRESS defining an abstract data model, predefined schemas defining lots of concrete data models for different application needs, the clear text encoding for file-based data exchange, and uniform data access independent of database technology through SDAI.

We have shown how to model and use FITS, a well-known scientific data exchange standard, in STEP. Especially the similar expressiveness of EXPRESS to modern programming languages allows for a quite seamless mapping of real world concepts. Some of its concepts like multi-class membership are even more powerful. Although (instance) methods may be simulated with derived attributes, we would prefer direct support to further improve application modeling. In addition, schema-central interface definition allows for a very clean mapping to most programming languages.

Furthermore, a solution has been proposed to optimize applications without changes to application code basically providing for early binding of schema information and optional, STEP-specific tuning descriptions (independent of DBMS); a code generator delivers optimized abstract data types (ADT) for entities. Our STEP-DBMS layer is portable to different ODBMSs and takes the corresponding requirements into account. The code generator provides high efficiency without sacrificing flexibility. Its extended ability to shift decisions before application run-time by binding them to STEP-specific data sets is ideally suited to support the two optimizing principles storage and caching with prefetching. Concluding, separating optional prefetching description from the (real) application program has the following advantage: schema and application development do not need to take into account proprietary optimizing principles of a given DBMS.

We have learned that SSDBM shares many requirements with CAD or product database management and, also, that there are specialties not directly supported by STEP. Professional SSDBM users should, therefore, participate in the standardization process to enforce support for their needs. There is, for example, no language binding for SDAI to FORTRAN due to lack of activity!

Acknowledgements

We would like to thank Dirk Wollscheid for his extensive implementation effort. We also thank Per Svensson for helpful comments and suggestions to improve this paper.

References

[1] M. Atkinson, R. Morrison: Orthogonally Persistent Object Systems. VLDB Journal, 1995, 4:319-401.

[2] F. Bancilhon, G. Ferran: The ODMG Standard for Object Databases. Proceedings 4th International Conference on Database Systems for Advanced Applications, DASFAA'95, pp. 273-283.

[3] R.G.G. Cattell: The Object Database Standard: ODMG-93 - Release 1.2. Morgan Kaufmann, 1996.

[4] A. Farris: Modeling Complex Astrophysics Data. Proceedings 7th International Working Conference on Scientific and Statistical Database Management, SSDBM'94, Charlottesville, Virginia, pp. 149-158.

[5] K. Geiger: Inside ODBC. Microsoft Press, 1995.

[6] GIDA mbH: Documentation for the EXPRESS Design and Programming Environment Version 3.0. Berlin, 1995.

[7] J. Gray: The Benchmark Handbook. Morgan Kaufmann, San Mateo, CA, 1993.

[8] A. Herbst: Long-Term Database Support for EXPRESS Data. Proceedings 7th International Working Conference on Scientific and Statistical Database Management, SSDBM'94, Charlottesville, Virginia, pp. 207-216.

[9] T. Härder, A. Reuter: Principles of Transaction Oriented Database Recovery, in: ACM Computing Surveys 15:4, 1983, pp. 287-317.

[10] T. Härder, T. Sauter, J. Thomas: Design and Architecture of the FDBS Prototype INFINITY. Submitted to International CAiSe'97 Workshop on Engineering Federated Database Systems, EFDBS'97, Barcelona, Spain, June 1997.

[11] ISO TC184/SC4/WG7: Product Data Representation and Exchange, Part 11: EXPRESS Language Reference Manual.

[12] ISO TC184/SC4/WG7: Product Data Representation and Exchange, Part 21: Clear Text Encoding of the Exchange Structure.

[13] ISO TC184/SC4/WG7: Product Data Representation and Exchange, Part 22: STEP Data Access Interface.

[14] ISO TC184/SC4/WG7 N403 (Committee Draft): Product Data Representation and Exchange, Part 23: C++ Programming Language Binding to the Standard Data Access Interface Specification.

[15] W. Kim: Object-Oriented Approach to Managing Statistical and Scientific Databases. Proceedings 5th International Working Conference on Statistical and Scientific Database Management, SSDBM'90, Charlotte, NC, pp. 1-13.

[16] D. Loffredo, M. Hardwick: Efficient Database Implementation of EXPRESS Information Models. 4th International

EXPRESS Users Group Conference, Greenville, SC, 1994. US Product Data Association (1995).

[17] H. Lührsen, T. Krebs: STEP Databases as Integration Platform for Concurrent Engineering. Proceedings 2nd International Conference on Concurrent Engineering, Johnstown, 1995, pp. 131-142.

[18] D. Maier, D.M. Hansen: Bambi Meets Godzilla: Object Databases for Scientific Computing. Proceedings 7th International Working Conference on Scientific and Statistical Database Management, SSDBM'94, Charlottesville, Virginia, pp. 176-184.

[19] J. Melton, A.R. Simon: The New SQL: A Complete Guide. Morgan Kaufmann Publishers, 1993.

[20] M. Maier, G. Staub: Object Modeling Technique (OMT) and EXPRESS: Comparison of "Two Worlds". Proceedings International EXPRESS User Group Conference, EUG'95, Grenoble, 1995.

[21] U. Nink: Efficiency of SDAI on top of ODBMSs (in german). Informatik Xpress 8, Proceedings CAD'96, Verteilte und intelligente CAD-Systeme, Kaiserslautern, Germany, March 1996, pp. 430-445.

[22] U. Nink, N. Ritter: Database Application Programming with Versioned Complex Objects. Informatik aktuell: Proceedings of the GI-Fachtagung "Datenbanksysteme in Büro, Technik und Wissenschaft", Ulm, Germany, March 1997.

[23] J. Owen: STEP: An Introduction. Information Geometers, 1993.

[24] G. Ozsoyoglu, Z.M. Ozsoyoglu, K. Vadaparty: A Scientific Database System for Polymers and Materials Engineering Needs. Proceedings 7th International Working Conference on Scientific and Statistical Database Management, SSDBM'94, Charlottesville, Virginia, pp. 138-148.

[25] M. Rafanelli: Research Topics in Statistical and Scientific Database Management. Proceedings 4th International Working Conference on Statistical and Scientific Database Management, SSDBM'88, Rome, Italy, pp. 1-18.

[26] B. Rieche, K.R. Dittrich: A Federated DBMS-Based Integrated Environment for Molecular Biology. Proceedings 7th International Working Conference on Scientific and Statistical Database Management, SSDBM'94, Charlottesville, Virginia, pp. 118-127.

[27] F. Schönefeld, C. Böhm: Using EXPRESS Database Technology for Accessing NCBI Genomic Data. 3rd International EXPRESS Users Group Conference, Berlin, Germany, 1993.

[28] A. Segev, A. Chatterjee: Supporting Statistics In Extensible Databases: A Case Study. Proceedings 7th International Working Conference on Scientific and Statistical Database Management, SSDBM'94, Charlottesville, Virginia, pp. 54-63.

[29] ISO/IEC 9075-3:1995: Information technology -- Database languages -- SQL -- Part 3: Call Level Interface.

[30] P. Svensson: Position Paper to panel: Are Commercially Available DBMS Good Enough? Proceedings 4th International Working Conference on Statistical and Scientific Database Management, SSDBM'88, Rome, Italy, pp. 388-398.

[31] J. Vuoskoski, M. Dach: Using EXPRESS in a High Energy Physics Research Environment. 4th International EXPRESS Users Group Conference, Greenville, SC, 1994. US Product Data Association (1995).

[32] W. Wilkes, T. Kretzberg: EXPRESS+ and SDAI+: Generating Application-Specific Programming Interfaces for Product Data Management (in german). Informatik Xpress 8, Proceedings CAD'96, Verteilte und intelligente CAD-Systeme, Kaiserslautern, Germany, March 1996, pp. 150-164.

WWW pages

[33] A. Farris: AIPS++: Astronomical Information Processing System.
http://www.cv.nrao.edu/fits/src/
http://aips2.nrao.edu/aips++/

[34] G. Heine: Product Description Standards.
http://www2.echo.lu/oii/en/products.html

[35] J. Nell: STEP on a Page. NIST.
http://www.nist.gov/sc5/soap/

[36] B.M. Schlesinger: FITS Basics and Information. Hughes STX.
http://www.gsfc.nasa.gov/astro/fits/basics.info.html

[37] I. Stern: Scientific Data Format Information FAQ.
http://www.cv.nrao.edu/fits/traffic/scidataformats/faq.html.