Data description and conversion architecture

by R. A. Demers
K. Yamaguchi

A data description and conversion architecture has been defined by IBM to enhance data interchange among Systems Application Architecture™ (SAA™) programming languages and systems. Its components, described in this paper, are (1) A Data Language (ADL), a programming language for describing data and specifying what data conversions are to be performed, (2) an object-oriented method of encoding ADL for efficient machine storage, transmission, and processing, and (3) programs that translate the data declarations of other programming languages to or from ADL. Also discussed is the application of the architecture to record-oriented files for SAA Distributed File Management.

Many different programming languages are in use today on a wide variety of computers and operating systems. Each programming language is best suited for a specific range of tasks and has language features and data types designed for those tasks. For example, FORTRAN was designed for scientific computing, COBOL for commercial data processing, and C for systems programming. But this is not to say that commercial programs cannot be written in FORTRAN, or scientific programs in C. In fact, there is considerable overlap of features and data types among programming languages, the result of their evolution and the adoption of features from one another.

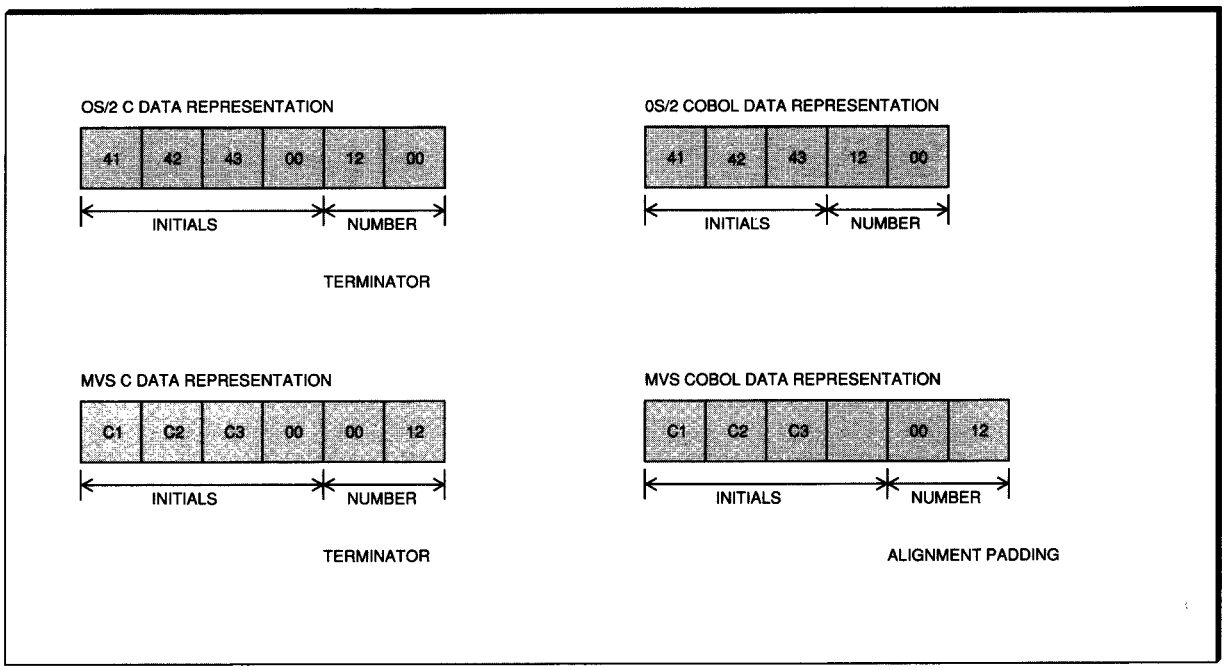
It is a cliché that programmers should use the best language for the task at hand, but this advice is often ignored. Programmers tend to use the languages that they know best or that are mandated by their employers. It is common to hear of "COBOL shops" or "C shops," for example. An important benefit of this standardization is that the programs of a shop specializing in one language can exchange data among themselves. For example, a COBOL program can *export* data by writing the data to a file, sending the data in a message, or calling another COBOL program and passing data as arguments. A second COBOL program can *import* data by reading the data from a file, receiving a message, or accepting arguments to its parameters. It is only necessary for the exporting and importing COBOL programs to use matching interfaces, consisting of the same COBOL data declarations and complementary COBOL export/import statements (WRITE/READ, SEND/RECEIVE, etc.).

But today, as business entities are continually reshuffled, applications are increasingly mixtures of different languages. And therein lies the problem; programming languages are not designed to communicate with each other. In particular, the data exported by programs of one language often cannot be used when imported into programs of other languages. The exporting and importing programs simply have no agreements regarding the type, representation, aggregation, and alignment of data.

When the exporting and importing programs are on different systems, the number of incompatibilities increases further. Each implementation of

©**Copyright** 1992 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to *republish* any other portion of this paper must be obtained from the Editor.

Figure 1 The diversity of data representation techniques



a language deals with these representation issues in the ways best suited for its target hardware and operating environment. The methods used often vary even for different implementations of the same language.

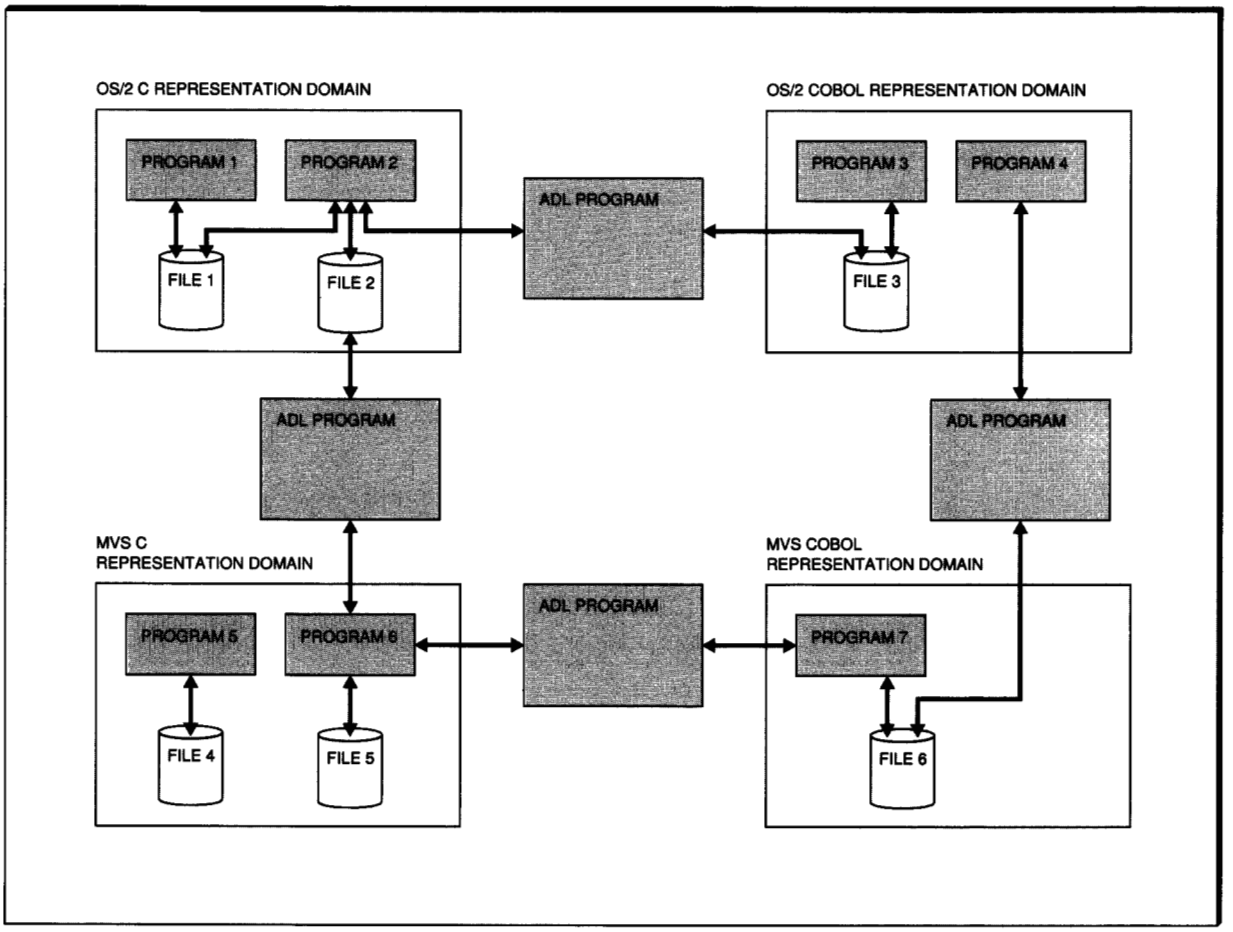
Figure 1 illustrates the diversity of methods used for a simple data structure. In this figure, a data structure called RECORD, consisting of a three-character field called INITIALS and a two-byte binary field called NUMBER, is shown as represented by C programs and by COBOL programs on Operating System/2* (OS/2*) systems and on Multiple Virtual Storage (MVS) systems. The character data are ASCII-encoded in the OS/2 systems but EBCDIC-encoded in the MVS systems. And, the data are null-terminated in C programs but not in COBOL programs. The binary data are byte-reversed in the OS/2 systems but not in the MVS systems. The binary field is aligned on a two-byte boundary in MVS COBOL but not in OS/2 COBOL. The diversity found in this simple example makes it easy to imagine the representation incompatibilities that would be found in more complex cases.

Data incompatibility problems can be handled by application logic, but this logic can be very difficult to program because of the numerous technical details to be considered. It would be far better if these incompatibilities could be overcome by conversion logic automatically invoked by system services as data are imported or exported.

This paper describes a data description and conversion architecture (called DD&C architecture) that has been defined by IBM for data interchange among Systems Application Architecture* (SAA*) programs. Designed for SAA Distributed File Management (DFM) products, DD&C has been applied to record-oriented file access and can be applied to data interchange through other mechanisms. As an architecture, DD&C is a set of specifications for the construction of products that implement the functions and interfaces defined by DD&C architecture. In this paper, we do not describe any particular products built to the DD&C specifications. It will be up to IBM and other software vendors to incorporate DD&C capabilities into products as they see fit.

A key concept of DD&C architecture is that data conversions must be performed whenever data flows between *representation domains*. The concept of representation domains is introduced by

Figure 2 ADL programs bridge between representation domains

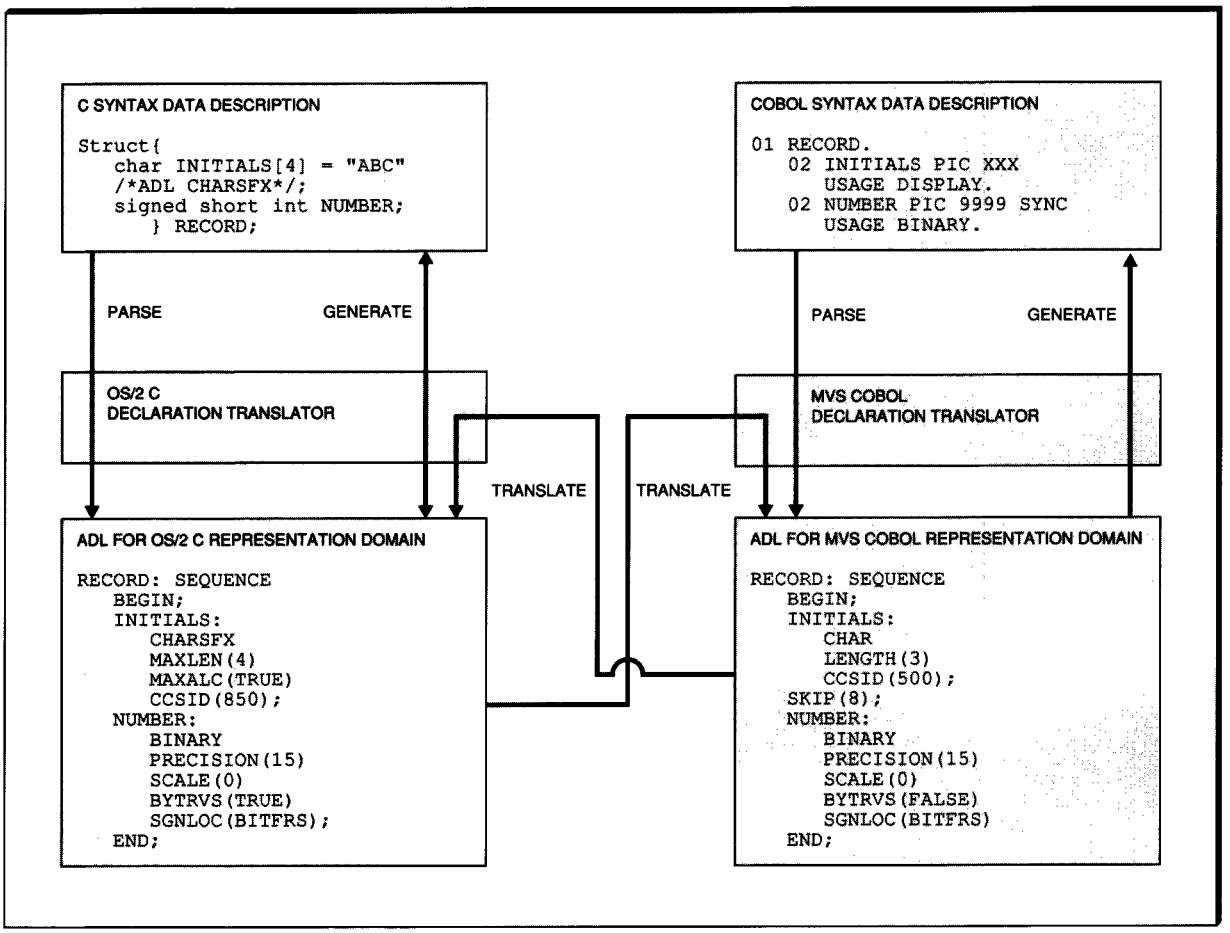


DD&C architecture to help categorize data representations in terms of the methods used by a particular language implementation. For example, OS/2 C, MVS C, OS/2 COBOL, and MVS COBOL are four different representation domains because of differences in data types between C and COBOL, and because of representation differences for the OS/2 and MVS systems. This arrangement is illustrated by Figure 2. The programs that perform necessary conversions are defined by a new programming language, called *A Data Language*, or *ADL*,[1] designed specifically for describing how data are represented by different representation domains and for converting data as the data flow between representation domains.

The top of Figure 3 shows the data declaration statements used by C and COBOL programs to describe the simple data structure illustrated in Figure 1. The C and COBOL declarations have little in common beyond the names that programmers used for the structure (RECORD) and its variables (INITIALS and NUMBER). It is not possible to compare the C and COBOL declarations to determine what conversions are needed as data flow between a C program and a COBOL program. Not only are there syntactic differences, neither declaration is complete in regard to data representation. Language data declarations are deliberately abstract so that programmers need not be concerned with details that can be more easily handled by language compilers.

In contrast, the ADL declarations at the bottom of Figure 3, corresponding to the C and COBOL declarations, are each complete for a particular rep-

Figure 3 Declaration translator functions

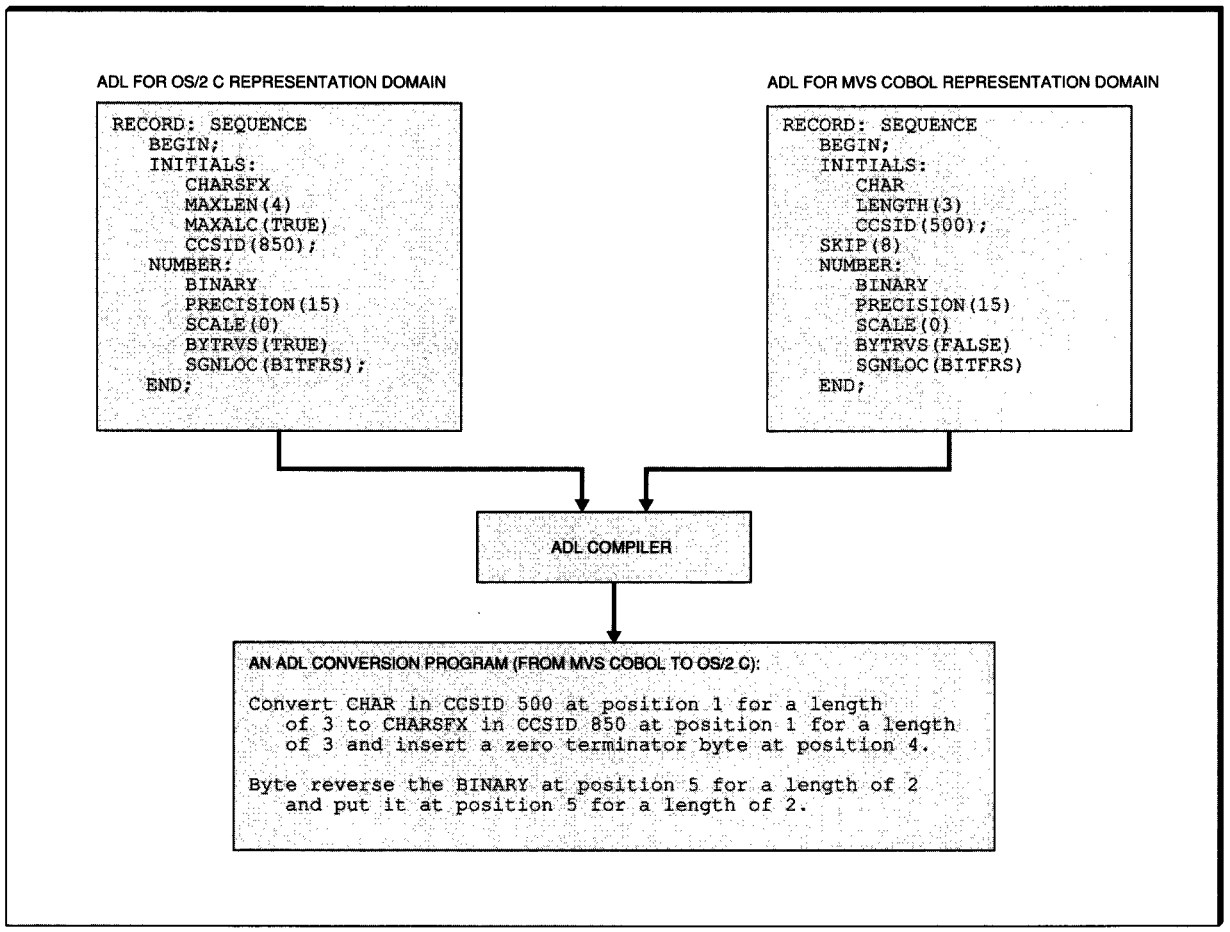


resentation domain because they fully describe the representations of the structure and its variables. They are comparable to each other because they are expressed in a common language—ADL. There are many differences between the ADL declarations for the OS/2 C and MVS COBOL representation domains, but clearly, an ADL compiler, as shown in Figure 4, can identify these differences and generate an appropriate conversion program. Then, when data are to flow from an MVS COBOL program to an OS/2 C program, as shown in Figure 5, the compiled ADL program can perform the necessary conversions.

Where do the ADL declarations come from? They can, of course, be written by a programmer, but that programmer would need to understand all of the details of both the data declarations of a language and the methods of representing data by a representation domain. Requiring this knowledge defeats the whole point of abstract data types in programming languages. Language compilers are already capable of handling all of these details, but compilers do not make this information available outside of themselves in a consistent and usable fashion. It would be practically impossible to make them do so. Some other mechanism is needed.

DD&C architecture defines a new kind of program called a *declaration translator*. As shown in Figure 3, a declaration translator for a particular representation domain can *parse* the data declarations of its associated programming language and create an ADL declaration with all representation

Figure 4 Creating conversion programs from ADL declarations



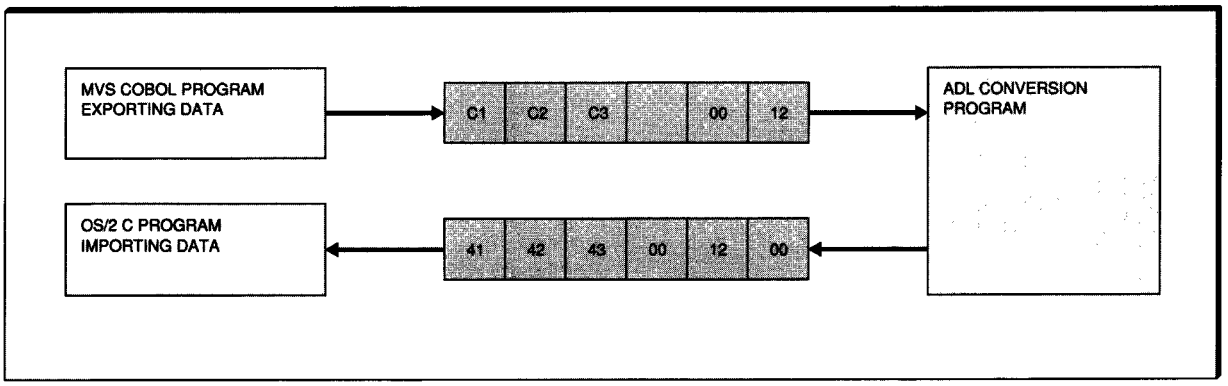
details filled in. A COBOL programmer needs only to pass valid COBOL data declarations to this declaration translator function.

A declaration translator can also reverse this process and *generate* a data declaration for its associated language from an ADL description that is correct for its representation domain. These language declarations can then be included in programs as part of their export-import interface. For example, a COBOL programmer can include COBOL data declarations generated from ADL in any COBOL program that needs to export or import the data described by an ADL declaration.

In addition, declaration translators can *translate* ADL descriptions appropriate to a different representation domain into an ADL appropriate to their own representation domain. For example, if there is an ADL description of data that was written to a file by an MVS COBOL program, the OS/2 C declaration translator can translate it into an ADL description correct for the OS/2 C representation domain. Together, these three declaration translator capabilities (parse, generate, and translate) provide a rich set of tools in support of multilanguage programming.

What remains is to decide when an ADL program needs to be called. This decision depends on the export-import mechanism being used. For example, as shown later in Figure 9, if an MVS COBOL program has written records to a file and an OS/2 C program needs to read them, the OS/2 READ service can call an ADL conversion program as each record is read. DD&C architecture is not con-

Figure 5 Converting data



cerned with why or when ADL programs are called, but IBM's Distributed File Management products deal with these issues for record-oriented files.

In the remainder of this paper, we consider the objectives of DD&C architecture, discuss the alternatives considered in its design, examine the major concepts of DD&C in more depth, and finally, discuss the application of DD&C to file access and to program calls.

Objectives

General objectives. The general objective of DD&C architecture is to enable programs written in any of the SAA programming languages (C, COBOL, CSP, FORTRAN, PL/I, and RPG), as implemented on any of the SAA system types, such as Multiple Virtual Storage/Enterprise Systems Architecture (MVS/ESA*), Virtual Machine/Enterprise Systems Architecture (VM/ESA*), Operating System/400* (OS/400*), and OS/2, to interchange data. This is not to say that all data exported by any of these programs can be imported by other programs of the set. There are entirely too many language and implementation peculiarities to allow for a fully general solution. For example, numeric data formatted as character strings can only be interchanged as character data; conversion to the numeric data types is not supported. Instead, DD&C architecture defines a practical solution that allows most SAA data to be interchanged by most SAA programs. Further, the initial design of DD&C architecture is focused on the interchange of data stored in record-oriented files.

Data description objectives. Innumerable record-oriented files, created by a wide range of applications, exist in midrange and large systems. Providing workstations with access to the data in this "data jail" is a prime objective of the SAA Distributed File Management products.[2] But the differences between the languages and data representations typically used in workstations and those used in midrange and large systems must be bridged to make this access feasible.

Programmers can write ADL descriptions of the data stored in files, but it can be quite difficult to do correctly because each implementation of a programming language has its own ways of representing, aligning, and aggregating data. Programmers would have to be fully aware of all of them. Even in the initial scope of DD&C architecture, ADL is quite complex and supports many options. Requiring programmers to learn and correctly use ADL would not be acceptable to them. Instead, an objective of DD&C architecture is to make use of existing data declarations specified in the syntax of an SAA programming language and provide automatic translation into ADL via a declaration translator for each SAA representation domain.

Data conversion objectives. The key objective of DD&C architecture in converting data can be stated in one word: performance. Data conversion is always viewed as a necessary evil, something required to get a job done, but which contributes nothing to the application itself. Therefore, DD&C architecture seeks to minimize

the amount of data conversion that must be performed and to perform it as efficiently as possible.

The initial focus of DD&C architecture on record-oriented files substantially affected its design. During file access, we are often dealing with a large number of records, each with a potentially large number of fields. Each record must be converted as efficiently as possible, with as little overhead as possible, and affecting as few fields of the record as possible.

The fundamental approach of DD&C architecture is to preplan the conversions that will be required for each record by comparing the descriptions of the records of the file with a description of how a program is designed to view those records. Differences in data representation, alignment, and aggregation can then be accommodated by a single invocation of an optimized conversion program for each record, performing only necessary conversions with minimal overhead.

A second data conversion objective is to have all conversions allowed by DD&C architecture produce the same results wherever these conversions are performed. For example, a conversion from IEEE (Institute of Electrical and Electronics Engineers) floating point to hexadecimal floating point should produce exactly the same result whether the conversion is performed on OS/2 or on MVS/ESA. To this end, the specifications for ADL carefully define the results to be produced for all conversions allowed by ADL.

A third data conversion objective of DD&C architecture is enhanced data interchange. Many differences in programming language data types exist for historical reasons but are of little importance to programs seeking to interchange data. For example, a number is a number is a number! The fact that one language represents numbers in binary format, whereas another represents them in zoned decimal format or floating-point format is often not relevant to an application. What is important, however, are the precision, scaling, and radix of the number. The ADL descriptions of data account for these differences, and ADL supports conversions among all numeric data types. ADL then goes one step further by ensuring that only data valid for the target data variable are actually assigned to it. Interchange is further supported by the translate function of declaration translators whereby each attempts to map any ADL-described data into the data types of its associated programming language.

A fourth data conversion objective of DD&C is support of application development. ADL is a programming language whose programs are invoked to convert data, but these conversions are not limited to simple data format conversions. Depending on how an application's view of the data is described, other conversions required by the application can also be performed. As examples, only certain records of a file can be selected, the fields of selected records can be reordered, and the fields of exported records can be validated.

And as a final data conversion objective, we must consider reversible conversions. If a program writes a record to a file of a different representation domain, that record must be converted to the representations required by the representation domain of the file. If the same program subsequently reads the same record from the file, it must be converted back to the representations of the representation domain of the program. Ideally, the record read should be identical to the record written, but this is not always the case. In numeric conversions, significant digits can be lost during the conversion because of rounding and truncation. In character conversions, best-fit character substitutions are used in some cases, with no best-fit on the return trip. Known solutions to these problems all involve canonical data formats that are incompatible with the other objectives of DD&C architecture, or require application programmers to account for reversibility in their designs.

Alternatives considered

Throughout the design of DD&C architecture, the architects were continually being asked questions such as "Why did you do X this way?" or "Why did you not use the Y architecture?" Indeed, many other alternatives were studied and considered. In this section, a few of the more important alternatives are discussed.

Programming language data description and conversion. Most programming languages (PLs) include facilities for describing the data contained in variables and for converting data as variables are assigned to other variables. Why are these PL descriptions and conversions not sufficient to allow data interchange among programs of different

representation domains? The short answer is that each PL implementation is self-contained with no consideration given in its design to data interchange with other PLs. A programmer must use matching interfaces of the same PL implementation when exporting data from the variables of a program and then importing data into the variables of another program.

Conversions of the representations used by different implementations of the same PL are possible if the associated PL data types are unaffected. So it would be acceptable, for example, for an OS/2 C program to access data stored in an MVS C file as long as representation conversions are somehow performed. However, the position is often taken that these conversions should be performed by implementations of the language that recognize when programs are being compiled for an environment consisting of heterogeneous systems. Using data descriptions embedded within program text, an approach to data conversion similar to that outlined in the later subsection on Network Interface Data Language then becomes feasible. After all, that approach was designed for a single language environment to begin with, that of C.

This unilanguage restriction is often eased to allow a certain amount of interlanguage data communications. But a requirement is imposed that the data types and interfaces of the exporting and importing PLs be "similar." This approach, in effect, calls for pair-wise agreements to be reached between PLs as to what types of data can be interchanged between those languages, agreements that have not yet been negotiated or documented for most PL pairs.

As an alternative to pair-wise agreements, programs that require interlanguage communications can be written within the constraints of a *common data model* (CDM), such as the one espoused by AD/Cycle*.[3] If only the data types of a PL that match the data types of the CDM are used in communications with other programs, interlanguage communications are assured. This alternative, however, has severe limitations. First, the data types of the CDM are, by necessity, the lowest common denominator of the data types found in the PLs covered by the CDM. Communications are restricted between any two PLs to the types held in common with the CDM, even when those PLs may well be able to interchange additional data types. Second, this alternative only applies to new programs and ignores the wealth of data locked up in the records of existing files, records that were written using the full expressive capabilities of some PL.

At the heart of this alternative is the view that the data types of a language are "owned" by that language and that only implementations of the PL should be allowed to operate on their instances. Any other operations on them could potentially result in bad data that cannot be processed by programs of the PL. But all that is really necessary is to understand the range of values that a variable can hold, as defined by its data type, and ensure that no value is passed to a program outside of that range.

Many existing alternatives were studied and considered throughout the design of the DD&C architecture.

If we look specifically at data stored in files, this alternative ignores historic realities. Most files residing in large or midrange systems were typically written by COBOL, FORTRAN, PL/I, or RPG programs. But the programs executing in workstations that seek access to the data in those files are typically written in C, BASIC, Pascal, or other languages. No pair-wise interlanguage data conversion agreements exist, and few compilers have been designed for a heterogeneous systems environment.

The use of PL data conversion capabilities when accessing files on heterogeneous systems was rejected as being unrealistic, impractical, and premised on an outdated view of data processing.

Sun XDR. Sun eXternal Data Representation (XDR)[4] is one of several methods of interchanging data based on the use of canonical data formats, that is, based on rules specifying how data are to be represented when the data are transmitted between systems. For example, when character data are to be transmitted, XDR specifies precisely

how the data are to be encoded, and again, when floating-point data are to be transmitted, XDR specifies how those data are to be encoded.

The sending program must convert each data item to be transmitted to the format required by XDR for its data type. These conversions are accomplished by calling special XDR subroutines for each data item. These subroutines convert the value of a program variable to the canonical format, if necessary, and copy it to the message. When all such data items have been copied, the program calls another XDR subroutine to transmit the message via network pipes.

Data interchange, data in existing files, and the need to accommodate multiple languages are important.

The receiving program issues an XDR call to receive the message and then, for each data item to be received, the program must issue an XDR call to a subroutine that extracts the next value from the message, converts it to local representation if necessary, and returns it to a program variable. It is the programmer's responsibility to issue the correct XDR calls in the correct order to properly receive each value.

Although conceptually simple, this approach also has some disadvantages:

- For each representation domain, a set of XDR subroutines must be provided that converts local data representations to or from canonical formats.
- If the local representations of data are not the same as the canonical representation, conversions must be performed, even if the sending and receiving representation domains use the same local representations.
- XDR provides no support for constructor data types, such as arrays or nested structures. The sending program must decompose the constructor into a series of simple data elements by using program logic, and the receiving program must reconstruct them via program logic.
- XDR provides no means of communicating metadata between the sending and receiving programs. Side information, in the form of program listings and specifications, must be passed between the programmers of the sending and receiving programs. There is no means, for example, of recording the description of the data in a file with the file itself so that it is available to all users of the file.
- XDR provides no means of performing structural conversions of the messages transmitted, such as reordering or omitting fields or whole messages. Structural conversions are an application responsibility.
- XDR applies primarily to new programs because it requires them to be written in a certain way. Using XDR for communications between existing programs would require substantial changes to be made to the sending and receiving programs.
- Finally, XDR cannot be made to apply to the data in existing files because those data are not in the XDR canonical format.

All in all, there is sufficient justification for looking at other alternatives.

OSI ASN.1. Abstract Syntax Notation 1 (ASN.1)[5] is a language defined by the Open Systems Interconnection (OSI) standard of the International Organization for Standardization for describing data transferred between heterogeneous systems. Along with ASN.1 are the OSI Basic Encoding Rules (BER)[6] that specify a canonical format in which the data described by ASN.1 can be transferred.

The text of ASN.1 data descriptions is written primarily as a way for one programmer to tell another how data are to be transmitted. It is up to the sending program to encode data as required by the BER, performing whatever data conversions are necessary. It is up to the receiving program to decode these data from the BER format and convert the data to local representations. In some cases, utility programs have been developed to interpret ASN.1 descriptions and convert data between local representations and BER. However, the ASN.1, being an abstract data description, requires additional (unstandardized) annotations in order to correctly describe local representations. That is, programmers must learn

how the PL they are using actually represents data, learn how to write ASN.1, and then learn how to annotate it to specify how the PL actually represents data.

The OSI BER specifies how data are to be represented when the data are transferred between systems. For each item of data, the length of the data and a tag corresponding to an identifier specified in ASN.1 are transmitted along with the canonical representation of the data. Although this approach is robust, it too has some clear disadvantages:

1. The length and tag associated with each data item are superfluous. These metadata are not needed because the sending and receiving programs must be written to process matching data anyway. They do allow conversions to be performed by the OSI Presentation Level, but not enough information is available to really match PL representation requirements. If the data being transferred consist of repetitions of the same formats, as the data do for the records of a file, there is no need to transmit metadata more than once.
2. The canonical representation formats of BER were designed for universal interpretability and do not match the representations actually used on any system. Therefore, all data must be converted twice, once to the canonical format and once from the canonical format, even if the sending and receiving representations are the same. Few applications can afford this overhead, so data must be transmitted as unencoded byte strings, with real conversions performed by applications as necessary.
3. The data stored in existing files are not encoded as required by BER, and ASN.1 is not capable of describing the data without extensive annotations.

Although other, more efficient, canonical interchange languages than BER are possible, the need for any such interchange format was questioned during the design of DD&C architecture. Instead, the approach in DD&C architecture is based on detected differences in data descriptions in a language capable of completely specifying how data are actually represented.

Network Interface Data Language. The Network Interface Data Language (NIDL) of the Open Software Foundation's Distributed Computing Environment[7] is another language for describing data transferred between heterogeneous systems. NIDL is actually C language data descriptions with annotations that specify descriptive information not expressible in pure C. Thus, there is a clear relationship between NIDL and at least one PL, thereby making available at least the data description capabilities of C. But of course, not all programs are written in C.

Associated with NIDL are the rules of network data representation (NDR) for multicanonical data representation. NDR effectively overcomes the problems of OSI BER. No length or tag fields are transmitted, and each data type can be represented in one of several ways. A single tag is transmitted once to specify which representation will be used for each data type, allowing the sending system to specify the representations used and requiring it to perform the minimum conversions. The target then only has to convert items that differ from its local representations. For example, if a receiving program expects an IEEE binary floating-point number but is informed that a hexadecimal floating-point number has been sent, it knows to perform that conversion.

The key problem with NIDL is its single-language focus. Not designed to accommodate the needs of multiple languages, NIDL is not rich enough to describe the full range of data types and representations actually used by communicating programs. Nor would its C-language syntax be acceptable to programmers of other languages, even if it could be enhanced to meet the data description requirements of other languages.

FD:OCA. Formatted Data:Object Content Architecture (FD:OCA)[8] is an IBM document content architecture that was designed as a means of describing tabular data included in documents. IBM's Distributed Relational Database Architecture (DRDA)[9] uses FD:OCA to describe database tables transmitted between systems. DD&C architecture initially attempted to enhance FD:OCA so that it could be used to encode ADL. This enhancement proved undesirable for two major reasons. First, the design of FD:OCA was not flexible enough to accommodate the changes required by ADL. Second, the performance objectives of DD&C architecture did not allow the overhead of building and parsing FD:OCA data streams.

Data description specifications. The System/38* and its successor, the Application System/400* (AS/400*), provide a way independent of programming language to describe the data stored in some of the record-oriented files of those systems. Compilers can request the data description specifications (DDS) from a file and generate appropriate data declarations within the programs that access a file. Programs compiled with common DDS data descriptions can easily interchange data, even if written in different programming languages.

There are two reasons why DDS works on these systems. First, the various IBM-supplied compilers for these systems have all been written to exist within a common language execution environment. Data representation is standardized for similar language data types. Second, DDS is restricted to describing records that are simple collections of fields. None of the constructor types of the programming languages, such as arrays or nested structures, can be described. Unfortunately, these factors do not prevail in general; there is no universal common language execution environment, and many existing files do contain constructor type data. In fact, System/38s and AS/400s usually contain many files that are not described by DDS and are not easily interchanged with other representation domains.

Consideration was given to extending DDS, but extension was rejected because of the rigid, forms-oriented nature of DDS. However, many aspects of the relationship of DDS to files were adopted by DD&C architecture (see the subsection "File Data Description"). AS/400 DDS is considered to be a separate representation domain and its requirements have been accommodated in the design of ADL.

Concepts of DD&C architecture

The conceptual layers of DD&C architecture are shown in Figure 6. These layers are not part of DD&C architecture, as such.

Foundations. The layer of DD&C architecture at the bottom of Figure 6, called foundations, is concerned with the ways in which data are represented, aligned, and aggregated by various representation domains. This layer includes three representations now described.

Programming language data representations. In assembly languages, program variables are described in terms of their representation in a computer system. Historically, this approach made sense. Hardware architects are primarily interested in the representation of data as part of the operational specifications of a system, and the designers of assembly languages are interested in making those data representations available to programmers. But the programmer is burdened with the tasks of selecting the best representation schemes for values and of mapping variables onto memory for optimal processing.

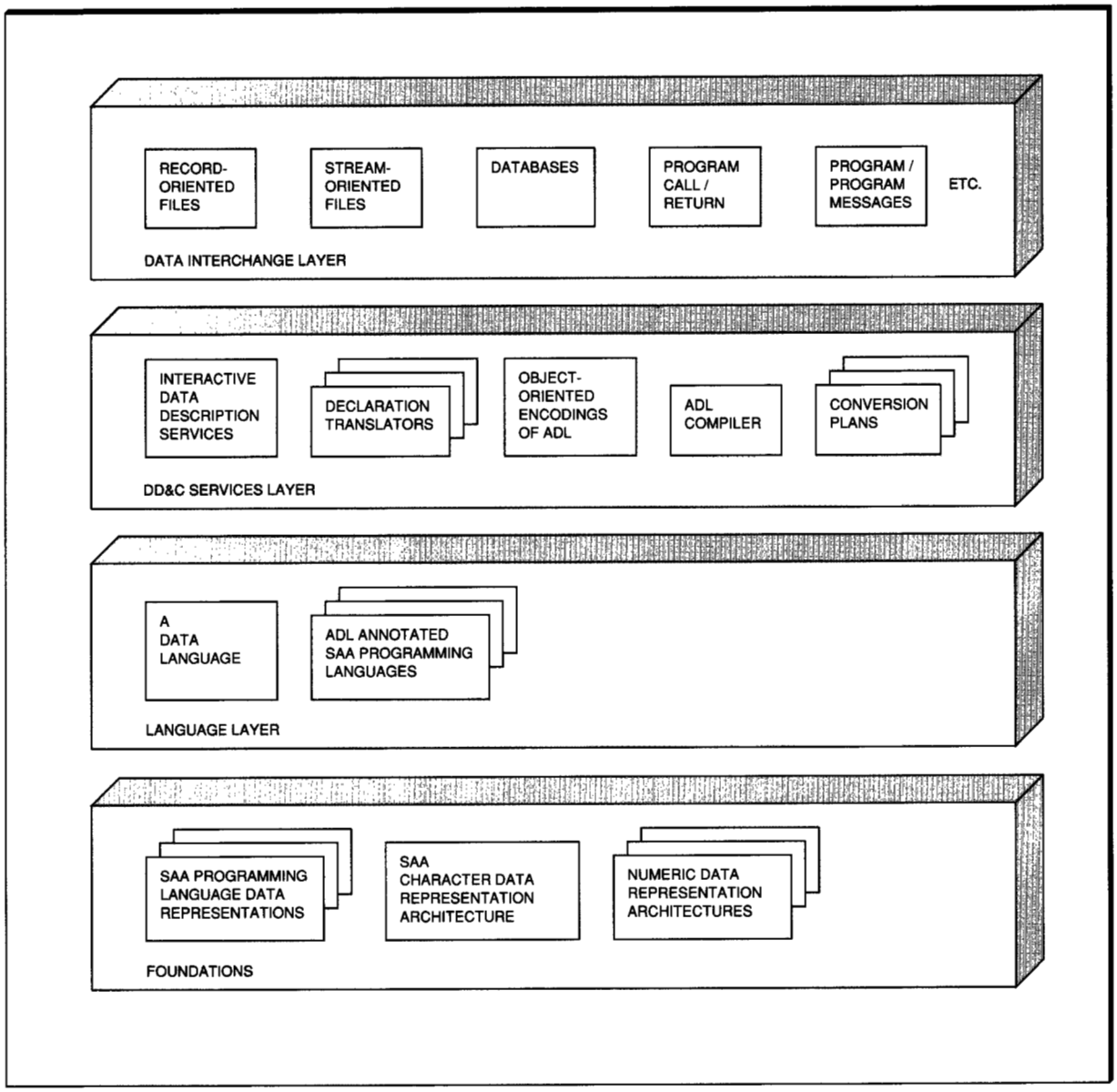
Programming language designers attempted to remove these burdens by defining a set of abstract data types that the programmer can use to describe variables. Each data type of a programming language defines a set of possible values and the operations that can be performed on those values. Examples of numeric data types are COBOL numeric PICTUREs, PL/I FIXED BINARY, and C integers. Numeric data types support arithmetic operations, and character data types support string operations. Programming languages support both simple data types concerned with single values and more complex data types, such as arrays and structures, concerned with aggregations of data.

For each of its data types, a programming language also defines a set of type attributes that can be specified by programmers to complete the description of a variable. For example, the PL/I FIXED BINARY data type also allows the programmer to specify the precision and scaling factor of the number, how the variable is to be initialized, and whether it is to be aligned or not.

The problem is that each programming language handles the representation, mapping, and attribute issues in its own way. Each compiler for a language is designed for a specific system architecture and its representation and mapping techniques. Thus, a COBOL program developed and tested on a personal computer (PC) may not work exactly the same as on a System/370* because of data representation differences. Data written to a file by a PC program cannot be directly read by the same program running on the System/370.

A difficult aspect of the design of DD&C architecture was deciding what representation domains

Figure 6 The conceptual layers of DD&C



should be formally considered in the scope of ADL. There was a strong temptation to make ADL as broad as possible, encompassing the representations used by as many IBM and other vendor systems as possible. However, this temptation was countered by the difficulty of concurrently defining ADL and a set of declaration translators. Experts could be found within IBM for work on declaration translators for each of the SAA programming languages and systems, but to go beyond this scope was not practical. We believe, however, that much of ADL is useful for other representation domains and that it is possible to formally extend DD&C architecture to other representation domains in the future.

Character Data Representation Architecture. A special case of data representation is that of char-

Figure 7 ADL program for file data conversions

```
(100)     FILE: /* declarations corresponding to file data */
                DECLARE
                BEGIN;
 (104)          RECORD: SEQUENCE
                        BEGIN;
                                INCLUDE 'DEFAULTS.MVS.COBOL';
  (104)                 INITIALS: CHAR LENGTH(3)
  | (105)                         WHEN INITIALS BETWEEN 'A' AND 'Z';
  (104)                 NUMBER:   BINARY PRECISION(15)
  | (105)                         WHEN NUMBER < 99;
                        END;
                END;

(100)     VIEW: /* declarations corresponding to program variables */
                DECLARE
                BEGIN;
 (104)          RECORD: SEQUENCE
                        BEGIN;
                                INCLUDE 'DEFAULTS.OS2.C';
  (104)                 INITIALS: CHARSFX MAXLEN(4)
  | (106)                         WHEN INITIALS IN ('A', 'B', 'C');
  (104)                 NUMBER:   BINARY PRECISION(15);
                        END;
                END;

(100)     ACCESS_METHOD: /* declarations of additional Access Method */
                         /* variables passed to ADL plans */
                DECLARE
                BEGIN;
 (101)          DEFAULT BINARY BYTRVS(FALSE) COMPLEX(FALSE) CONSTRAINED(FALSE)
                               RADIX(2) SCALE(0) SIGNED(TRUE) FIT(ROUND);
 (102)          word:      CONSTANT 31;
 (103)          ccsid:     SUBTYPE OF BINARY PRECISION(32) SIGNED(FALSE);
 (104)          inlen:     BINARY PRECISION(word);
                inccsid:   ccsid;
                outmaxlen: BINARY PRECISION(word);
                outccsid:  ccsid;
                outlen:    BINARY PRECISION(word);
                END;


(200)     getPlan: /* Plan to convert records read from the file */
                   PLAN (
                         inlen:       INPUT,
 (201)                   inccsid:     INPUT,
 (201) (203)             FILE.RECORD: INPUT LENGTH(inlen) CCSID(inccsid),
 (201)                   outmaxlen:   INPUT,
 (201)                   outccsid:    INPUT,
 (202) (204)             VIEW.RECORD: OUTPUT MAXLEN(outmaxlen) CCSID(outccsid),
 (202)                   outlen:      OUTPUT)
                      BEGIN;
 (205)                   VIEW.RECORD <- FILE.RECORD;
 (205)                   outlen <- LENGTH(VIEW.RECORD);
                      END;

(200)     putPlan: /* Plan to convert records to be written to the file */
                   PLAN (
 (201)                   inlen:       INPUT
 (201)                   inccsid:     INPUT
 (201) (203)             VIEW.RECORD: INPUT LENGTH(inlen) CCSID(inccsid),
 (201)                   outmaxlen:   INPUT
 (201)                   outccsid:    INPUT,
 (202) (204)             FILE.RECORD: OUTPUT MAXLEN(outmaxlen) CCSID(outccsid),
 (202)                   outlen:      OUTPUT)
                      BEGIN;
 (205)                   FILE.RECORD <- VIEW.RECORD;
 (205)                   outlen <- LENGTH(FILE.RECORD);
                      END;
```

acter data. A large number of different schemes have been developed, because the number of different graphic characters that can be represented by a single byte is limited (a maximum of 256 characters). Many national languages require special characters (e.g., accented vowels or punctuation) or entirely unique sets of graphic characters (e.g., Farsi). Some programming languages require special characters (e.g., APL). Some disciplines require special characters (e.g., mathematics). Various devices have been designed to support only limited graphic characters (e.g., printers and displays).

Different standards groups have promulgated a variety of encoding schemes (e.g., ASCII versus EBCDIC). Additionally, double-byte encoding schemes have been adopted for national languages requiring thousands of graphic characters (e.g., Chinese, Japanese, and Korean), along with a number of schemes for intermixing single-byte and double-byte encodings.

The result is a nightmare of complexity that greatly inhibits data communication. Fortunately, this complexity has been addressed by the IBM SAA Character Data Representation Architecture (CDRA).[10] CDRA defines a comprehensive means of identifying the scheme by which a given character string has been encoded. A single *tag* called a coded character set identifier (CCSID) can be associated with each character string. CDRA also defines the set of all possible valid conversions of character strings from one CCSID to other CCSIDs and provides the conversion tables required to perform them.

ADL associates CCSID tags with character strings, as required by CDRA, and depends on CDRA conversion services for performing character conversions.

Numeric representation architectures. The representation of numeric values falls into two major categories, the representation of exact numbers and the representation of approximate numbers. Again for a variety of historic reasons, a number of representation methods are in common use in each of these categories.

Exact numbers can be encoded in either binary or decimal format. Binary formats are often used because they are natively supported by most machine architectures and can therefore be processed most efficiently. Decimal formats exist because they are a better match to the ways in which numbers were encoded in the punched card devices of early systems. Not all decimal values can be accurately encoded in binary encodings because of inherent number system differences.

Approximate numbers are based on the concepts of scientific notation, which allow very small or very large numbers to be encoded with only a limited amount of precision. Based on exponential notation, various *floating-point* encoding schemes have been devised, using either binary or hexadecimal encodings of the characteristic and mantissa of the value.

Conversions between exact and approximate encodings of a numeric value are often possible but may result in data loss through truncation or rounding. ADL defines comprehensive rules for performing conversions of numeric encodings.

Language layer. The language layer is concerned with ADL and its relationship to the SAA programming languages.

A Data Language. ADL provides a means of describing the representation of data exported and imported by a wide variety of programming languages, as implemented on a variety of systems, so that efficient data conversion programs can be generated.

A module is the compilation unit of ADL, consisting of a DECLARE statement for each view of the data and PLAN statements for each required conversion program. See Figure 7 and Figure 8 for examples of ADL modules. Parenthesized numbers, such as (100) in the following text, refer to portions of those figures.

Each DECLARE statement (100) corresponds to a single representation domain and consists of:

- DEFAULT statements (101) that each specify the default values to be used for the attributes of a single ADL data type. These statements can be included from a library containing the DEFAULT statements for each representation domain, greatly reducing the number of details that must be specified in data declaration statements.
- CONSTANT statements (102) that associate a literal with an identifier. The identifier can then be referenced wherever the literal can be specified.

Figure 8 ADL module for program call conversions

```
(100)     PGMA: /* declarations corresponding to the OS/2 COBOL compiler's
                   representations of PGMA program variables */
                DECLARE
                BEGIN;
  (104)         INITIALS: CHAR LENGTH(3) CCSID(850) JUSTIFY(LEFT) UNITLEN(8);
  (104)         NUMBER:   BINARY PRECISION(3) BYTRVS(TRUE) COMPLEX(FALSE)
                                 CONSTRAINED(TRUE) RADIX(10) SCALE(0)
                                 SIGNED(TRUE) FIT(ROUND);
                END;

(100)     PGMB: /* declarations corresponding to the OS/2 C compiler's
                   representations of PGMB program variables */
                DECLARE
                BEGIN;
  (104)         INITIALS: CHARSFX MAXLEN(4) CCSID(850) MAXALC(TRUE) SFXENC(X'00'
                                  UNITLEN(8);
  (104)         NUMBER:   BINARY PRECISION(15) BYTRVS(TRUE) COMPLEX(FALSE)
                                 CONSTRAINED(TRUE) RADIX(2) SCALE(0)
                                 SIGNED(TRUE) FIT(ROUND);
                END;

(200)     gluePlan: /* Plan to convert records read from the file */
                    PLAN (
  (201)                  PGMA.INITIALS: INPUT,
  (202)                  PGMA.NUMBER:   OUTPUT)
                    BEGIN;
  (205)                  PGMB.INITIALS <- PGMA.INITIALS;
  (206)                  CALL 'PGMB' (PGMB.INITIALS, PGMB.NUMBER);
  (205)                  PGMA.NUMBER <- PGMB.NUMBER;
                    END;
```

- SUBTYPE statements (103) that define subtypes of the ADL-defined types, or of another subtype. Subtypes can be defined for each of the data types of a programming language, with the subtype identifier identical to a programming language type. For example, an *integer* subtype of the ADL BINARY type can be defined to be equivalent to the *integer* type of C. A set of SUBTYPE statements in a file, along with appropriate DEFAULT and CONSTANT statements, can ease the task of creating ADL descriptions of data if no declaration translator is available for a programming language.

  Subtypes can also be defined for application types. For example, a *gameboard* subtype of the ADL ARRAY type could be defined for use by all 8-by-8 game programs. These statements can also be included from a dictionary of common data descriptions.

- Data declaration statements (104). These statements actually describe data. Each consists of a keyword that names a data type, followed by attributes appropriate to the data type. For constructor types, a clause follows that defines the elements of the constructor, such as the elements of an ARRAY.

  A WHEN clause can also be specified in data declaration statements. In any assignment of data to the declared variable, the predicate expression[11] of the WHEN clause must evaluate to true or else the assignment statement and the plan are terminated with an exception. For files, two primary uses are seen for this capability: data selection and data validation. A WHEN clause (105) in the declaration of a file ensures that only valid data are written to the file. A WHEN clause (106) in the declaration of a program selects the records to be presented to the program.

Each PLAN statement (200) defines the parameters that can be passed to a conversion program as INPUT parameters (201) or as OUTPUT param-

Table 1 Scalar data types supported by ADL

ASIS	A bit string of unknown encoding. Assignment to any other ADL data type is allowed. When an ASIS field is the data source, the assumption is made that ASIS data are already represented as required by the target field and can be copied to it. When an ASIS field is the assignment target, the bit representation of the source field is simply copied to the ASIS target.
BINARY	A binary encoded number. Conversions to or from any other ADL numeric data type are defined.
BIT	A bit string. Conversions to or from the ADL BITPRE data type are defined.
BITPRE	A bit string with a length prefix. Conversions to or from the ADL BIT data type are defined.
BOOLEAN	An encoding of TRUE or FALSE.
CHAR	A string of characters encoded as specified by the associated coded character set identifier attribute. Conversions to or from any other ADL character data type are defined.
CHARPRE	A character string with a length prefix. Conversions to or from any other ADL character data type are defined.
CHARSFX	A zero terminated character string. Conversions to or from any other ADL character data type are defined.
ENUMERATION	An association of identifiers with integers. Conversions between ENUMERATION fields are performed such that the identifiers associated with the BINARY encodings of the ENUMERATIONs are preserved. Numeric conversions to or from the ADL BINARY, PACKED, and ZONED data types are also defined.
FLOAT	A floating-point number. An attribute specifies which IBM hexadecimal format (single, double, or extended) or IEEE format (single, double, or extended) is used. Conversions among all formats are defined. Conversions to or from any other ADL numeric data type are defined.
PACKED	A packed decimal number. Conversions to or from any other ADL numeric data type are defined.
ZONED	A zoned decimal number. Conversions to or from any other ADL numeric data type are defined.

eters (202). Each parameter is specified by the identifier of a data item declared in one of the DECLARE sections of the module. The parameter is thereby associated with an expected representation.

For file conversion programs, a single record is passed as an input parameter (203) and a single area is passed as an output parameter (204) to receive the converted record. Other input parameters specify the length and CCSID of the input record and the maximum length and required CCSID of the output record.

The parameter lists of these conversion programs are fixed in format because file conversion programs are called by access method services. But for other applications, such as CALL/RETURN conversions between programs, other parameters can be specified, as in Figure 8.

The ASSIGNMENT statements (205) of a PLAN request conversions as data are copied from a source variable to a target variable. The conversions allowed by ADL are defined by a matrix of conversions from one ADL type to other ADL types. In addition to scalar type conversions, ADL includes powerful conversions of constructor types that greatly simplify the writing of conversion plans. In fact, default plans are defined for most file data conversions, so that programmers need only be concerned with describing the data in files, and not with writing ADL plans.

An ADL PLAN can also CALL (206 in Figure 8) other programs and pass and receive parameter values. These programs can be called for several reasons. One reason is to perform conversions not otherwise supported by ADL. Another reason is to allow an ADL PLAN to act as a *glue program* between otherwise incompatible programs, as shown in Figure 8. A third reason is to allow an ADL PLAN to perform conversions as part of a larger function, such as a remote procedure call. All ADL PLANs and all programs called by ADL PLANs conform to the program calling conventions established by the SAA AD/Cycle Common Execution Environment.[3]

The scalar data types supported by ADL are listed in Table 1.

Attributes of each type specify metadata that describe how an instance of a type is represented

and used. For the BINARY scalar type, for example, the following attributes can be specified: PRECISION, SCALE, RADIX, SIGNED, COMPLEX, CONSTRAINED, LENGTH, BYTRVS, FIT, TITLE, HELP, and NOTE. Some of these attributes, such as TITLE and HELP, can be optionally specified, but a value must be specified for other attributes in a data declaration statement or through inheritance from a SUBTYPE statement or a DEFAULT statement.

The constructor data types supported by ADL are the following:

- ARRAY—a unidimensional or multidimensional collection of elements. The bounds of each dimension can be specified by a low bound and either a high bound or size. Declaration translators can map between programming languages that specify array bounds in either way. The bounds can be any signed integer as long as the low bound is less than the high bound. Further, the bounds of each dimension can be specified by integers, constants, or by reference to other integer fields passed in a record or as parameters.

  The elements of an ARRAY can be of any scalar or constructor type or subtype except ARRAY. ADL requires arrays of arrays to be described as a single multidimensional array, which is easily accomplished by the declaration translators of languages that do not support multidimensional arrays directly (such as COBOL).

  When an array is assigned to another array, the number of dimensions and the number of elements in each dimension must conform in the source (exported) and target (importing) arrays named in the assignment statement. Dimensionality is preserved because each dimension has semantic meaning to applications. However, the index values associated with each dimension for each element of the array are not preserved if the low bounds of a dimension differ. Some programming languages allow negative low bounds, whereas others require the low bound to be 0 or 1. The ADL goal of data interchange takes precedence over preservation of dimension cardinality.

- CASE—a declaration of a multiformat data element, for example, when the records of a file can be of several formats. A CASE declaration consists of a set of WHEN statements and an optional OTHERWISE statement. The predicate expressions of the WHEN statements are evaluated in the order specified until one evaluates to true. The data declaration associated with that WHEN statement is then selected. If no WHEN statement evaluates true, the data declaration of the optional OTHERWISE statement is selected.

  For files, it is expected that predicate expressions will be used to evaluate *discriminator* fields within the records to determine their format. But in other cases, the discriminator can be passed to the PLAN as a separate parameter; for example, when converting the messages of a mapped conversation, the map name can be passed to the PLAN as a parameter.

  The selected data declaration of a WHEN statement can be of any scalar or constructor type or subtype.

  When a CASE is assigned to another CASE, the WHEN clauses of the source are first evaluated to identify the source format. The data declaration of the target WHEN statement with a matching identifier is then selected as the target format. If the assignment statement completes successfully, the predicate expressions of the selected target WHEN statements are evaluated to ensure that the target format is valid for its discriminators.

- SEQUENCE—an ordered collection of data items, each with its own declaration. The elements of a SEQUENCE can be of any scalar or constructor type or subtype.

  When a SEQUENCE is assigned to another SEQUENCE, the identifier of each declaration of the target sequence is matched to the identifiers of the source sequence. When a match is found, the corresponding source declaration is selected for assignment to the target. The target can be a subset of the source, and the data items of the target sequence can be in a different order than those of the source sequence.

The ability to declare constructors whose elements are constructors is a powerful feature of ADL. This ability allows, for example, the declaration of ARRAYs of SEQUENCEs, SEQUENCEs of SEQUENCEs, one or more CASEs or ARRAYs within

a SEQUENCE, or ARRAYs of CASEs. For example, it is possible to describe an array in which the format of each element is unique.

ADL annotation of programming languages. Although ADL text can be written to fully describe data exported by the SAA programming languages (PLs), a DD&C objective is to use existing descriptions specified in the syntax of the SAA PLs to the extent possible. In many cases, PL text provides sufficient information to allow an ADL description be produced without any further consideration by programmers. However, additional information is sometimes required, especially if data are to be exported to other PLs or imported from other PLs. Although it would be desirable to enhance the PLs so that the additional information could be specified natively in each PL, this expectation is not realistic, given the rigidity of language standards. Instead, this information can be provided as ADL annotation to PL text.

ADL annotation of programming language text provides additional information for data that will be exported or imported.

A general model of ADL annotation of PL text was designed by the developers of the declaration translators for the SAA PLs. Seeing common problems across the PLs led them to generalizations about what kinds of annotation are required and how they can be incorporated in PL text. A description of the general annotation model is beyond the scope of this paper, but some examples of its use will illustrate its power.

In general, ADL annotation consists of special comments intermixed with standard PL text to provide additional information about data. The first nonblank symbol after the opening comment symbol of the PL comment is the symbol "ADL." Following this symbol is text consisting of ADL statements or segments of ADL statements. This ADL annotation can be intermixed with PL text in whatever way is considered optimal by the declaration translator designers for each PL. Some simple examples of ADL annotation are the following:

- ADL includes certain optional attributes that affect neither the representation nor the conversion of data. Examples are TITLE, HELP, or NOTE. They only provide programmer or application commentary. These attributes can be specified in ADL annotation of PL variable declarations. In the following PL/I text, the ADL TITLE attribute is specified as an ADL annotation comment within a PL/I declaration statement.

```
DCL num
    /* ADL TITLE('Employee serial number')*/
    FIXED BINARY(31);
```

  PL/I compilers ignore the comment, but a PL/I declaration translator uses it to add the specified TITLE attribute to its ADL declaration of *num*:

```
num: BINARY PRECISION(31) TITLE('Employee
    serial number');
```

  A declaration translator generate function reverses this process, making the TITLE attribute an ADL annotation comment.

- ADL annotations are also used when a PL declaration can be mapped to ADL in more than one way, depending on the programmer's intended use. For example, a C char declaration would normally be mapped by a C declaration translator to the ADL CHARSFX (character with a suffix) data type, but the C language also allows a char variable to be used as an integer. If it is the programmer's intent to actually treat it as an integer, the following ADL annotation can indicate it is being used as a single byte signed binary count field:

```
signed char checked_out_books
    /* ADL BINARY; */;
```

  The C declaration translator, being familiar with the peculiarities of the C language, is then able to create the following ADL:

```
checked_out_books : BINARY PRECISION(7)
    SIGNED(TRUE);
```

- Complex numbers are important for scientific applications but are not supported by commercial programming languages, such as COBOL. If it is necessary to import a complex number into a COBOL program, the real and imaginary portions of the complex number must be declared as separate variables within a containing COBOL structure, as in:

```
*ADL D_06: COMPLEX(TRUE);
     15 D-06.
        20 REAL      PIC S99V99 COMP-4.
        20 IMAGINARY PIC S99V99 COMP-4.
```

  An ADL annotation comment precedes the COBOL structure declaration to specify that the structure is really just a single complex number. The COBOL declaration translator, being familiar with this convention, is able to create the following ADL that allows correct ADL conversion programs to be constructed:

```
D_06: BINARY COMPLEX(TRUE) LENGTH(16)
    PRECISION(4) SCALE(2) RADIX(10);
```

The concurrent design of ADL and of the SAA declaration translators allowed many such considerations to be discussed and appropriate changes made to both ADL and the declaration translators. Consideration was given to all three declaration translator functions (parse, generate, and translate).

Services layer. The services layer is concerned with the services that must be provided when implementing DD&C architecture.

Interactive data description. One of the objectives of DD&C architecture is to make it as painless as possible for programmers to describe data for ADL conversion programs. As previously discussed, having programmers manually write ADL data declarations would not be acceptable to them. Instead, DD&C architecture defines tools called declaration translators that allow them to use existing PL data declarations. But to be used properly, these tools should be used within the context of a computer-aided software engineering (CASE) environment that knows how PL data declarations are stored, how to use declaration translator functions, and how to relate ADL declarations to various interchange mechanisms, such as files or program calls.

Relating DD&C services to CASE environments is properly the task of CASE vendors and is outside the scope of DD&C architecture.

Declaration translators. As was described earlier in the introductory section, associated with each representation domain is the software component called a declaration translator. It understands the following things about its representation domain:

- The syntax used by programmers for describing data in the PL of that domain
- The data types supported by the PL
- The attributes of the PL used to qualify the data types
- The representations used by the compiler for each data type
- The method used by the compiler for aligning data in memory
- The methods used by the compiler for mapping constructor types to memory

Declaration translators provide programmers with the following three functions for using PL data descriptions:

1. PARSE—parses a PL data description to produce an ADL data description. This function fills in the details of the encodings used by the representation domain for each of the data types of the PL.

   The result of PARSE is an encoded form of ADL. An ADL declaration translator allows programmers to describe data in ADL itself. This capability is provided for PLs for which no declaration translator is otherwise provided.

2. GENERATE—generates PL text from an ADL data description. This function produces PL data descriptions that can be included in programs, thereby facilitating program development. ADL annotation is included as required to allow a subsequent invocation of the PARSE function of the representation domain.

   The input to GENERATE is an encoded form of ADL. The GENERATE of the ADL declaration translator can be used to produce ADL text for any representation domain, a valuable debugging aid.

3. TRANSLATE—translates the ADL description of data originally created for any other represen-

tation domain to produce an ADL description of data suitable for its own representation domain. This function helps programmers to make productive use of data exported by other representation domains by examining the encoding details of the other representation domain and filling in the encoding details of the target representation domain. The input and output of TRANSLATE are an encoded form of ADL.

Referring to the ADL declarations in Figure 3, the translation performed by the OS/2 C declaration translator on the ADL data declaration for MVS COBOL is the following:

- The declaration of RECORD as a SEQUENCE is copied because this is identical to the ADL used to describe C structures.
- A declaration of character data for OS/2 C data differs from that of MVS COBOL in both data type and attributes. Since C character strings are typically null-terminated, INITIALS must be described in ADL as an ADL CHARSFX data type. MAXLEN(4) is specified to allow for the null-terminator byte plus the three bytes of character data. MAXALC(TRUE) is specified because the CHARSFX field does not need to vary in length. And CCSID(850) is specified because OS/2 C character data are ASCII-encoded.
- The SKIP(8) specification of the ADL declaration for MVS COBOL is ignored because the OS/2 C declaration does not require pad bits to force alignment of the NUMBER field.
- The declaration of NUMBER is copied, but the BYTRVS attribute is changed to TRUE to reflect how short integers are encoded by OS/2 C.

Declaration translator transformations, plus additional editing of programming language text, provide great flexibility. By using ADL and an appropriate set of declaration translators, interlanguage communications between programmers and between tools, as well as conversions of data between native representations, become relatively easy. Programmers can use their normal languages and tools. They have to learn how to annotate PL descriptions with ADL, but it is only a small difference from their existing PL knowledge base.

The following interfaces for requesting declaration translator functions have also been defined:

- List declaration translators
- List declaration translator attributes
- Parse PL source text file
- Generate PL source text file
- Translate descriptor

As a final point regarding declaration translators, we believe that they should be implemented in a highly portable fashion so that they can be used wherever application programming is done. It would make little sense to separately implement each declaration translator for individual systems because of the cost and because of the inevitable inconsistencies that would result. Further, we expect a great deal of sharing of internal components and logic among declaration translators. Although there must be a separate declaration translator for each COBOL representation domain, for example, clearly there is much in common among all COBOL declaration translators. For both of these reasons—portability and commonality of components—an object-oriented design with a common class library would make sense for declaration translator implementations.

Object-oriented encoding of ADL. The syntax of ADL was designed for human reading and writing. It consists of free-format textual tokens that must obey the rules of ADL grammar while giving programmers considerable freedom of expression. As with other programming languages, ADL trades off processing efficiency for programmer ease of use.

The obvious solution to this conflict is to process ADL text and generate an encoded form of the language that is more suitable for storing, processing, and transmission. In fact, this step is often first in the compilation process for many programming languages. For ADL, however, the encoded form is not just an internal form of the language, being of no concern outside of each implementation. The encoded form of ADL is the primary means of communication among a set of software engineering tools residing on one or more heterogeneous systems. Among these tools are the ADL compiler (as shown in Figure 4) and declaration translators.

The design requirements for the ADL encodings were the following:

- Isomorphism with ADL. The encodings of ADL are just another form of the language. Anything that can be expressed in ADL can be expressed

in its encodings, and all relationships specified in ADL are maintained in the encodings. This led to the adoption of object-oriented technology in the encodings. The objects of the encodings are instances of classes that model each aspect of the language. For example, instances of the ADLBINARY class represent descriptions of variables of the ADL BINARY data type.

- Processing efficiency. The ADL encodings are directly processable, without any need for parsing or other forms of analysis. Any indirection allowed by ADL, such as named references to variables, are fully resolved in the ADL encodings. Optimization information that can be inferred from ADL, such as field lengths and offsets, is explicitly encoded.
- Extendability. ADL covers a large number of representation domains, but many additional languages and systems exist that were not considered in the initial design of ADL. Further, ADL was designed to meet only the requirements of IBM SAA Distributed File Management. It is anticipated that additional data types, attributes, and features will be required in ADL as additional representation domains and areas of application are considered. Therefore, the method of encoding ADL also allows for extensions in all of the directions in which ADL is likely to be extended. This requirement effectively rules out control-block forms of encoding and points to the use of linked-object structures.
- Transmission efficiency. The efficiency with which a data structure can be transmitted between systems is affected by its size and by the transformations required to linearize it as a data stream. Most ADL descriptions are relatively small, so size was not considered an important factor. The primary concern, here, was the design of a data stream architecture that linearized the linked-object structure selected for ADL encodings. ADL encoding objects are created within a single space-type object, with all relationships among them specified by offsets from the beginning of the space. In this way, the entire space can be transmitted and received without any further processing of its contents by the sender or the receiver.
- Malleability. Various aspects of an ADL data description are subject to change over time, including when a description is initially being created and when it is subsequently modified as application requirements change. To allow this level of malleability, it must be possible to add, delete, and modify the objects of the space that encode an ADL description. All objects of the encoding (except for an anchor object) are independent of their location in the space.
- Integrity. As changes are made to an ADL description, errors introduced by failing tools become a possibility. These errors can be avoided by allowing only proven tools, invoked through well-defined interfaces, to make changes. Encapsulation can be enforced for the space containing the description and for each class of object contained within the space. It further justifies the use of object-oriented programming techniques in the encoding of ADL.
- Storage efficiency. As with transmission efficiency, the size of a data structure is only one aspect of storage efficiency. In a paged memory environment, locality of reference is of great importance. That is, objects that are frequently used together (especially small objects) should be in close proximity in the space, whereas large, infrequently used objects should be stored where they do not interfere with normal reference patterns. The location independence of objects in a space is the prime requirement for this rule.

The encoded forms of ADL are specified and controlled through formal architecture specifications that can be published and thereby made available to the developers of CASE tools.

The ADL compiler. An ADL compiler consists of two components, a parser capable of converting ADL source text into encoded objects and a program generator that creates conversion programs for each specified plan of a module. Unlike most other compilers, these DD&C architecture components are separate. The ADL parser is actually the parse function of an ADL declaration translator, and what is called the ADL compiler is actually just the program generator. This division makes sense in DD&C architecture for several reasons:

- A full ADL declaration translator was needed anyway to support representation domains without their own declaration translators.
- Any number of declaration translators can be created that produce the object encoding of ADL as the output of their parse function.
- It is possible for other CASE tools to generate or use the information in the object encodings of ADL.

- The object encodings of ADL for files are stored with the files and can be used in multiple ADL modules.
- In Distributed File Management, it is occasionally necessary to regenerate conversion programs on the fly, so it is desirable to avoid the parse phases.

As with declaration translators, it is likely that ADL compilers will be needed on many different types of systems, and the programs produced by them must convert data identically on all systems. These are strong arguments for the development of a single highly portable compiler. And since the inputs to the compiler consist of encoded objects, an object-oriented compiler design would also be appropriate.

Conversion programs. The output of the ADL compiler is a program that can be called to perform data conversions. Consideration was given to both interpreted programs and directly executable programs. Initially, interpreted programs were favored because they could be easily transmitted between Distributed File Management systems. The system best able to perform conversions could then be dynamically selected to do them. However, this method was rejected because it led to considerable complexity in the Distributed Data Management (DDM) architecture[12] flows defined for Distributed File Management, because the dynamic selection of the conversion system proved less important than initially thought, and because the performance of interpreted programs is generally inferior to that of directly executed code.

Further, we believe that it is important for ADL compilers to generate as much in-line conversion logic as possible. Run-time subroutines will be used for many conversions, especially complex numeric conversions and CDRA character conversions, but in general, in-line logic should be used because it allows more optimization to be done. This is especially true for predicate logic, for the conversions of constructors, and for simple scalar conversions.

Data interchange layer. The data interchange layer is a conceptual layer, concerned with how application programs are developed for interchange and how ADL conversion programs are created and invoked. The Distributed File Management answers to these questions for record-oriented files are discussed, but similar considerations apply to other interchange mechanisms.

The following two subsections describe how file access DD&C is used by SAA Distributed File Management for the record-oriented files. For more information on SAA Distributed File Management, see Reference 13.

It is important for ADL compilers to generate in-line conversion logic because it allows more optimization to be done.

File data description. Record-oriented file systems evolved as repositories of data produced and used by programs written in a variety of programming languages. Each programming language includes interfaces for reading, writing, and updating records. When matching interfaces are used, the records written to a file by one program can be read by another program of the same language. These interfaces consist of programming language statements for accessing records, such as READ and WRITE statements, and statements for describing the data contained in the records.

The declaration statements can be specified within the source text of a program, but for important files, they are often stored in a library and dynamically included in the program by the programming language compiler. This method ensures that all programs that include the declaration statements share a common definition of the records of the file. However, it is not a complete solution. First, it works only for programs of a single programming language. Second, if any change is made to the stored data declarations for any one program, all programs that include it must be recompiled. Without adequate support tools, it can be difficult to determine which programs need to be recompiled, especially when the file is accessed by programs on remote systems. Third, there is no managed relationship between the data declarations and the file they describe. Even if all of the programs that access a file in-

cluded the same data declarations, nothing guarantees that the file actually matches those descriptions.

These problems are solved by the System/38 and its successor, the AS/400, for their common language environment. Language-independent, *external* data descriptions, called data description specifications (described earlier in this paper), are created by programmers during application development. These descriptions are stored with a file when the file is created. Compilers can request these descriptions from a file and generate appropriate data declarations within programs that access the file.

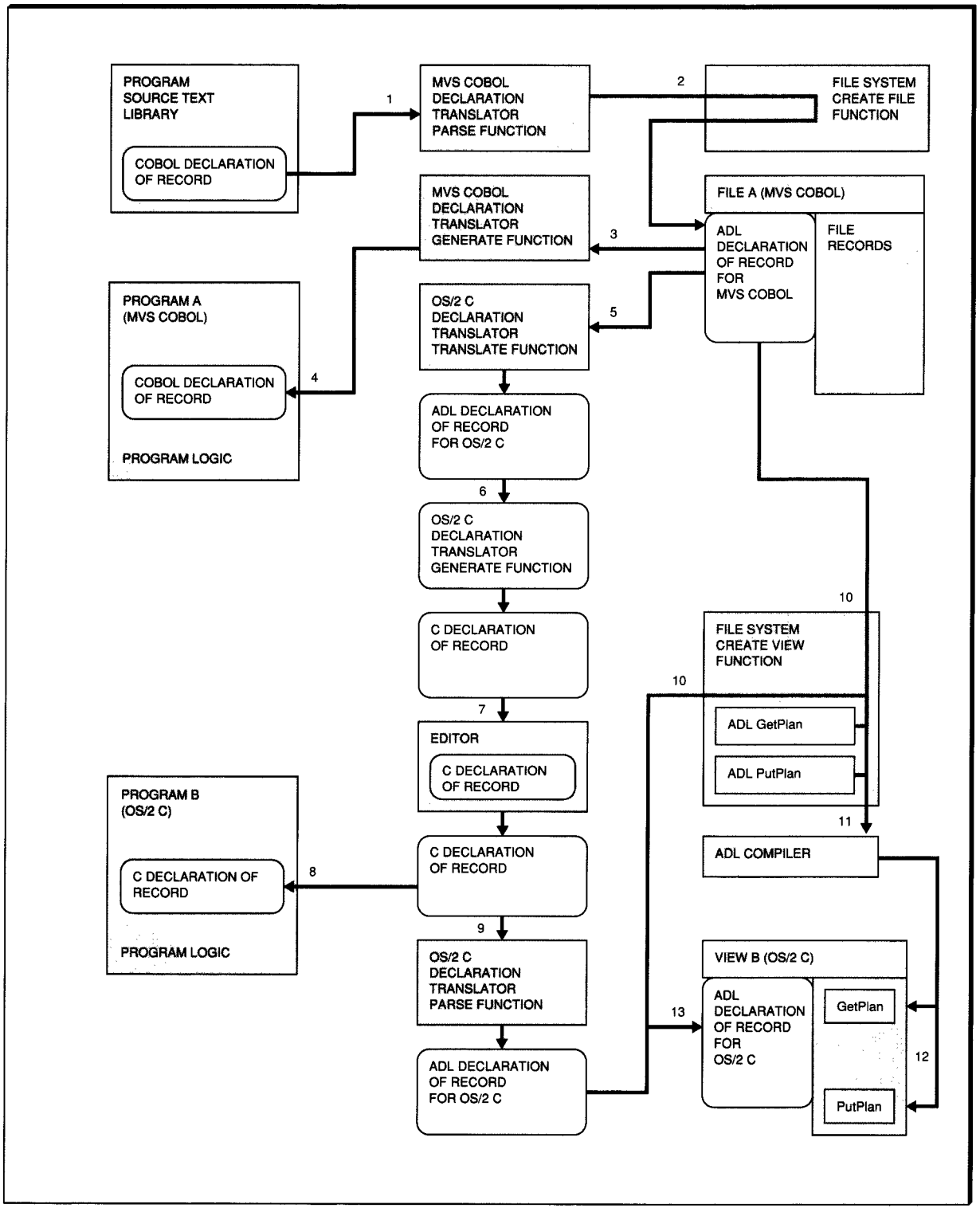
Another concept of the AS/400 pertinent to DD&C is that of *logical files*. A logical file is one that contains no records of its own, but instead provides an alternate view of the data in a *physical file*. In the alternate view, the format of the records, their representation, or their order can differ from those of the base physical file. If a program needs the data view or access path defined by the logical file, it opens the logical file. Any operations requested on logical file records are actually performed on the base physical file.

VIEW files are the means by which ADL conversion programs are managed and used for files. A view file is similar to an AS/400 logical file in that a program opens a view file when it needs data conversion services to use a base file. Conversion programs produced by the ADL compiler are stored with the view file and called as needed to convert records. View files can be managed the same as any other files in a file system with regard to naming, directory services, security, and locking.

These concepts of view files and of language-independent data descriptions stored with files are central to the DD&C approach to file data interchange. This approach is illustrated by Figure 9, with the following steps keyed to that figure. Note that these steps are assumed to occur within the context of CASE tools that guide and aid programmers in performing them.

1. When a file is created, an ADL description of its records can be specified as an attribute of the file. This description is specified in terms of the representation domain of the primary set of programs that will work with the file, MVS COBOL in this example. The COBOL declaration of the records is obtained from a library of program source text and passed to the parse function of the MVS COBOL declaration translator.
2. The resulting ADL description is passed to the create file function of the file system to be stored as an attribute of the file.
3. When it is time to compile a program of the same representation domain as the file (an MVS COBOL program), the ADL description stored with the file is requested and passed to the generate function of the MVS COBOL declaration translator.
4. The resulting COBOL declarations can then be included in the COBOL program and compiled. Although it would be possible to use the original COBOL declarations from the source text library, it is better to use the declarations generated from the ADL description of the file. The source library declarations could have been changed and therefore be out of synchronization with the file.
5. When it is time to compile a program from a different representation domain (an OS/2 C program in this example), the ADL declaration stored with the file can be used to create appropriate declarations for the new program. The ADL declaration of the file is passed to the translate function of the OS/2 C declaration translator.
6. The resulting ADL declarations can then be passed to the generate function of the OS/2 C declaration translator to produce C declarations.
7. The C declarations can be edited by a programmer as needed for the program being developed. For example, the programmer could reverse the order of the NUMBER and INITIALS fields, change the declaration of the NUMBER field from short integer to integer, change the CCSID (in the ADL annotation) of the INITIALS field, delete one of the field declarations, or add WHEN clauses to field declarations (as ADL annotation) to select the records of the file to be processed by the OS/2 C program. Although this example is very simple, a variety of changes would be possible in more complex cases to allow the new program to work with the data as it requires. However, the fewer changes made from the declaration produced by the translate function of the declaration translator, the easier it is to keep data declarations up to date with

Figure 9 Application development process



the original COBOL declaration of the records of the file.
8. The edited C declarations can then be included in a new C program and compiled.
9. The edited C declaration can also be passed to the parse function of the OS/2 C declaration translator to produce an updated ADL description of the program's view of the data.
10. The final task is to create a view file that can be used by the OS/2 C program to access the records in the MVS COBOL file. The ADL declaration stored with the MVS COBOL file and the new ADL declaration for the OS/2 C program's view of the records of the file are both passed to the create view function of the file system. Default ADL PLANs for converting records read from the file (similar to the GetPlan in Figure 7) and for converting records written to the file (similar to the PutPlan in Figure 7) are provided by the file system.
11. The ADL compiler is called by the create view function and passed the ADL declarations and plans.
12. The resulting GetPlan and PutPlan conversion programs are stored with the new view file for later use during data access.
13. The ADL description of the OS/2 C view is stored as an attribute of the view file so that it can be used by other OS/2 C programs (via step 3, above).

File data conversion. Figure 10 shows how data are converted by ADL conversion programs as records are accessed through a view file with the following steps keyed to the figure.

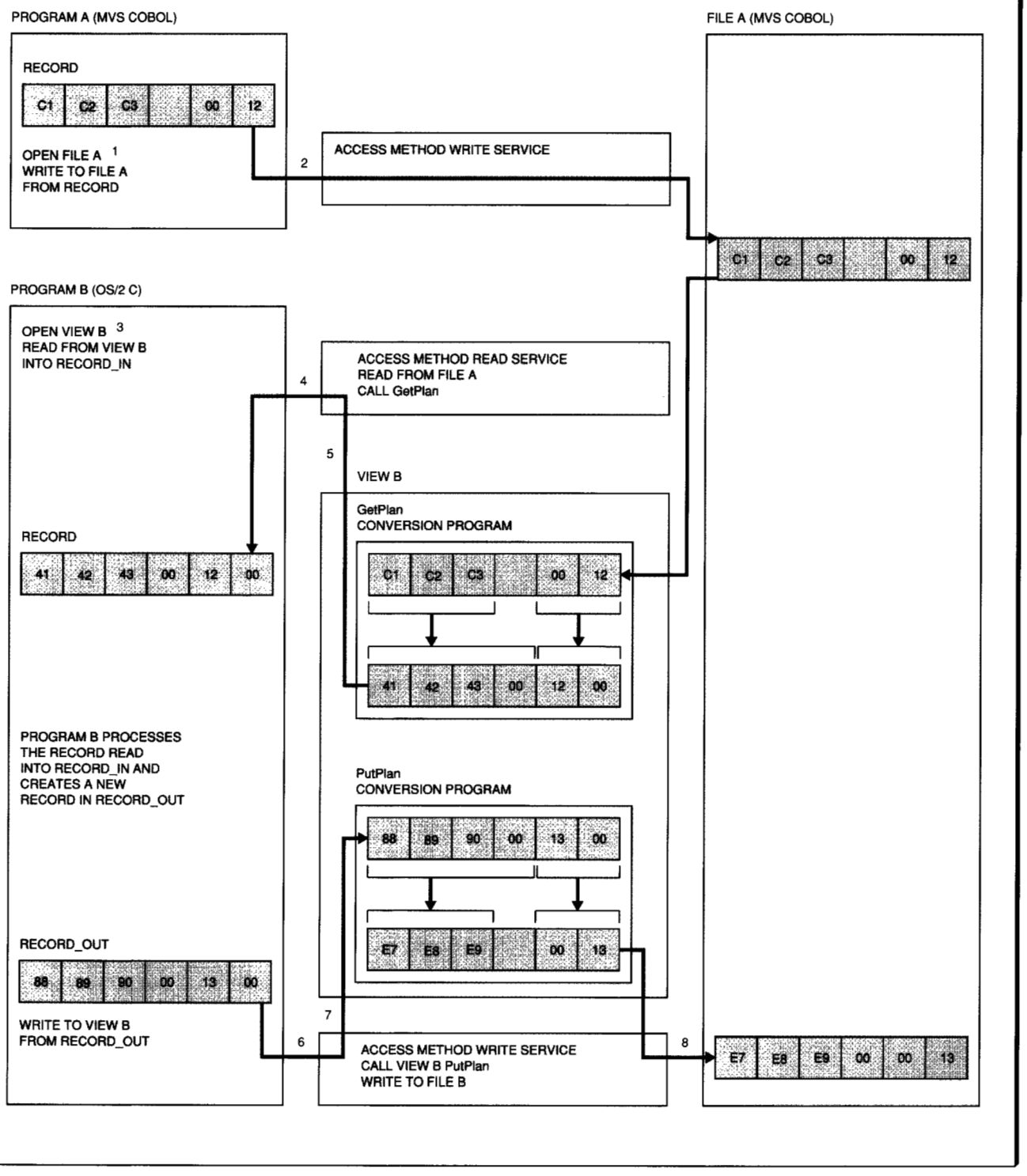
1. When a program opens a file by name, the file system creates a bound path to the file that its access method services can use to efficiently work with the records of the file. Since an MVS COBOL program is opening a file that is also in the MVS COBOL representation domain, there is no need for conversion services.
2. The MVS COBOL program can assign values to the fields of RECORD in its process memory and then use the COBOL WRITE statement to call access method write services to add the record to the file. Note that the record is written to the file essentially as it was mapped to process memory by the MVS COBOL compiler.
3. When a program opens a view file by name, the file system creates a bound path through the view file to its base file. Since an OS/2 C program needs to access an MVS COBOL file, a view file must be opened to provide the necessary conversion services.
4. When the OS/2 C program reads records from the view file, the access method read service obtains the record from the base file and calls the GetPlan of the view file.
5. The GetPlan converts the record to the form required by the OS/2 C program. The converted record is returned to the access method and is then presented to the OS/2 C program.
6. When the OS/2 C program writes a record to the view file, the access method write service calls the PutPlan of the view file.
7. The PutPlan converts the record to the form required by the MVS COBOL base file and returns it to the access method.
8. The access method writes the converted record to the base file.

Program call DD&C. Although DD&C architecture has been designed for data interchange through files, the need for data conversion support when using other data interchange mechanisms is readily apparent. Without going to the depths of explanation provided for files, the application of DD&C to program calls is illustrated in Figure 8. An ADL conversion program is used as an intermediary when a program of one representation domain (OS/2 COBOL) calls a program of a different representation domain (OS/2 C).

The ADL module consists of the following:

- The first DECLARE statement describes the variables of the OS/2 COBOL program being passed as arguments to the OS/2 C program.
- The second DECLARE statement describes the variables of the OS/2 C program that are defined as its parameters.
- The PLAN statement, named gluePlan, must be called by PGMA, instead of PGMB. The input parameter, INITIALS, is converted to the representation required by PGMB, and space is allocated for the output argument returned by PGMB, NUMBER. The gluePlan calls PGMB, passing it the converted INITIALS argument and the address of the space for the returned NUMBER argument. On return from PGMB, the value of NUMBER returned is converted to the representation required by PGMA, and the gluePlan terminates. If PGMB had failed, any exceptions would have been forwarded to PGMA from PGMB.

Figure 10 File data conversions



The primary issues to be resolved are the kinds of programming issues discussed for files in the subsection "File Data Description," namely how the declarations of interfaces are to be stored, translated, and managed. These issues are best considered within the context of a CASE system. As with files, considerable benefit can be derived from declaration translator parse, generate, and translate functions.

The concept shown in this example can be easily extended to interlanguage remote procedure calls.

Concluding remarks

Within its initial scope of the SAA languages and systems, DD&C architecture enhances the interchange of data stored in record-oriented files. In the design of DD&C architecture, equal consideration was given to the problems of describing data and of converting the data. Initially designed for record-oriented files, DD&C architecture can also be applied to other areas of data interchange.

Acknowledgments

In addition to the authors, many other people participated in the design of the DD&C architecture, among them Marsha Brown, Bijan Katebini, Steven Osborne, Elaine Patry, William Remay, and Kenneth Sissors of IBM Manassas; Susan Carswell-Hurdis, David Cole, Brian Cromwell, Scott Gallimore, Joseph Mesa, and Ejuana Vasquez of IBM Cary; Paul Arevalo and Takahsi Hasegawa of IBM Toronto; Langdon Beeck and Robert Moyer of IBM Santa Teresa; Lorenzo Falcon, Thomas Frayne, William Nettles, and Felix Nunez of IBM San Jose; Heinz Graalfs of IBM Sindelfingen; James Diephuis, Jan David Fisher, Sunil Gaitonde, James Wacholz, and David Weber of IBM Rochester; and Gregory Adams, James McGugan, and David Thomas of Object Technology International, Inc., Ottawa, Canada.

*Trademark or registered trademark of International Business Machines Corporation.

Cited references and notes

1. *Distributed Data Management Architecture: Specifications for A Data Language Level 1*, SC21-8286, IBM Corporation; available through IBM branch offices.
2. The SAA Distributed File Management products provide access to and management of distributed, record-oriented files, as defined by IBM's Distributed Data Management architecture (DDM). For more information on DDM, see Reference 12.
3. V. J. Mercurio, B. F. Meyers, A. M. Nisbet, and G. Radin, "AD/Cycle Strategy and Architecture," *IBM Systems Journal* **29**, No. 2, 170–188 (1990).
4. *External Data Representation Protocol Specification*, Sun Microsystems, Inc., Mountain View, CA (February 1986).
5. *Information Processing—Open Systems Interconnection—Specification of Abstract Syntax Notation One (ASN.1)*, ISO/DIS 8824, International Organization for Standardization, Geneva.
6. *Information Processing—Open Systems Interconnection—Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)*, ISO/DIS 8825, International Organization for Standardization, Geneva.
7. A. Hinxman and A. Simmons, *Functional Specification of the NIDL Language*, Digital Equipment Corp., Maynard, MA (1989).
8. *Formatted Data: Object Content Architecture Reference,* SC31-6806, IBM Corporation; available through IBM branch offices.
9. *Distributed Relational Database Architecture Reference* SC26-4651, IBM Corporation; available through IBM branch offices.
10. *Character Data Representation Architecture Level 1, Reference,* SC09-1390, IBM Corporation; available through IBM branch offices.
11. ADL predicate expressions are syntactically the same as those of Structured Query Language (SQL). Comparison predicates and the BETWEEN, LIKE, and IN predicates can be specified in expressions with Boolean operators. In ADL, however, they are used in several language constructs and do not have the set selection semantics of SQL predicates.
12. R. A. Demers, J. D. Fisher, S. S. Gaitonde, and R. R. Sanders, "Inside IBM's Distributed Data Management Architecture," *IBM Systems Journal* **31**, No. 3, 459–487 (1992, this issue).
13. R. A. Demers, "Distributed Files for SAA," *IBM Systems Journal* **27**, No. 3, 348–361 (1988).

Accepted for publication March 10, 1992.

Richard A. Demers *110 Salem Point SW, Rochester, Minnesota 55902.* Mr. Demers is a consultant on software architecture. In 1968 he joined IBM as an applications programmer in White Plains, New York. He received a B.A. degree in philosophy from Canisius College in 1969. In 1972, he moved to Endicott, New York, where he worked on systems software for the IBM 3895 Optical Check Reader. Mr. Demers moved to Rochester in 1975 to design the message handling and service components of the System/38 operating system. From 1982 to 1991, he was the lead architect in the design of IBM's Distributed Data Management (DDM) architecture, participated in the design of IBM's Distributed Relational Database architecture (DRDA), and was the lead architect for IBM's data description and conversion architecture. He has received IBM Outstanding Innovation Awards for his work on DDM architecture (1987) and DRDA (1991). A member of the Association for Computing Machinery, his professional interests include operating systems, distributed processing, data management, programming languages, and object-oriented programming.

K. (Koko) Yamaguchi *IBM Advanced Storage and Retrieval (ADSTAR), 5600 Cottle Road, San Jose, California 95193.* Dr. Yamaguchi is a senior programmer in the architecture and strategy group in the IBM San Jose ADSTAR programming laboratory. He joined IBM at Palo Alto, California, in 1974. Since then he has held numerous technical and managerial positions in the areas of high-level language development, internal tools development, and technical strategy. In 1988, he joined the San Jose programming laboratory to define and develop architectures and strategies for distributed structures. Dr. Yamaguchi received his Ph.D. from the University of Michigan in industrial and operations engineering in 1975. His major field of research was automated information system development.

Reprint Order No. G321-5483.