A SELF-DESCRIBING DATA TRANSFER METHODOLOGY FOR ITS APPLICATIONS

WA-RD 463.1

Research Report
December 2000

Washington State
Department of Transportation

Washington State Transportation Commission
Planning and Programming Service Center
in cooperation with the U.S. Department of Transportation
Federal Highway Administration

Research Report
Research Project T9903, Task 30
IVHS—Network and Data Fusion

A SELF-DESCRIBING DATA TRANSFER METHODOLOGY FOR ITS APPLICATIONS

by

D.J. Dailey, D. Meyers, and L. Pond
ITS Research Program
College of Engineering, Box 352500
University of Washington
Seattle, Washington 98195-2500

Washington State Transportation Center (TRAC)
University of Washington, Box 354802
University District Building
1107 NE 45th Street, Suite 535
Seattle, Washington 98105-4631

Washington State Department of Transportation
Technical Monitor
Dave Peach

Prepared for

Washington State Transportation Commission
Department of Transportation
and in cooperation with
U.S. Department of Transportation
Federal Highway Administration

December 2000

TECHNICAL REPORT STANDARD TITLE PAGE

1. REPORT NO. WA-RD 463.1	2. GOVERNMENT ACCESSION NO.	3. RECIPIENT'S CATALOG NO.
4. TITLE AND SUBTITLE A Self-Describing Data Transfer Methodology for ITS Applications		5. REPORT DATE December 2000
		6. PERFORMING ORGANIZATION CODE
7. AUTHOR(S) D.J. Dailey, D. Meyers, L. Pond		8. PERFORMING ORGANIZATION REPORT NO.
9. PERFORMING ORGANIZATION NAME AND ADDRESS Washington State Transportation Center (TRAC) University of Washington, Bx 354802 University District Building; 1107 NE 45th Street, Suite 535 Seattle, Washington 98105-4631		10. WORK UNIT NO.
		11. CONTRACT OR GRANT NO. Agreement T9903, Task 30
12. SPONSORING AGENCY NAME AND ADDRESS Washington State Department of Transportation Transportation Building, MS 7370 Olympia, Washington 98504-7370 Project Manager: Gary Ray, 360-705-7975		13. TYPE OF REPORT AND PERIOD COVERED Research report
		14. SPONSORING AGENCY CODE

15. SUPPLEMENTARY NOTES

This study was conducted in cooperation with the U.S. Department of Transportation, Federal Highway Administration.

16. ABSTRACT

A wide variety of remote sensors used in Intelligent Transportation Systems (ITS) applications (loops, probe vehicles, radar, cameras) has created a need for general methods by which data can be shared among agencies and users who own disparate computer systems.

In this paper, we present a methodology that demonstrates that it is possible to create, encode, and decode a self-describing data stream using the following:

1. existing data description language standards
2. parsers to enforce language compliance
3. a simple content language that flows out of the data description language
4. architecture neutral encoders and decoders based on ASN.1.

17. KEY WORDS Self-describing data, SQL, ITS data, ITS architecture	18. DISTRIBUTION STATEMENT No restrictions. This document is available to the public through the National Technical Information Service, Springfield, VA 22616

19. SECURITY CLASSIF. (of this report)	20. SECURITY CLASSIF. (of this page)	21. NO. OF PAGES	22. PRICE
None	None		

Disclaimer

The contents of this report reflect the views of the authors, who are responsible for the facts and accuracy of the data presented herein. The contents do not necessarily reflect the official views or policies of the Washington State Transportation Commission, Department of Transportation, or the Federal Highway Administration. This report does not constitute a standard, specification, or regulation.

Abstract

The wide variety of remote sensors used in Intelligent Transportation Systems (ITS) applications (loops, probe vehicles, radar, cameras, etc.) has created a need for general methods by which data can be shared among agencies and users who own disparate computer systems.

In this paper, we present a methodology that demonstrates that it is possible to create, encode, and decode a self-describing data stream using the following:

1. existing data description language standards,
2. parsers to enforce language compliance,
3. a simple content language that flows out of the data description language, and
4. architecture neutral encoders and decoders based on ASN.1.

Table of Contents

List of Figures

1. Self-Describing Data Overview

1.1 Introduction

The wide variety of remote sensors used in Intelligent Transportation Systems (ITS) applications (loops, probe vehicles, radar, cameras, etc.) has created a need for general methods by which data can be shared among agencies and users who own disparate computer systems. Such data sharing requires that both the sender and the recipient of the data agree on a method for transfer. To date, most systems constructed for this purpose (1) lack in generality or (2) are limited to data transfers of a specific type [1, 2, 3] or (3) are so general and complex as to be very difficult to implement [4]. The work presented in this paper was aimed at creating a general mechanism for *self-describing* data transfers of data streams that are produced by a set of remote sensors that change in number and type as a function of time. We present our self-describing data transfer concept in the context of Intelligent Transport Systems applications; however, our approach is applicable to a variety of data types and sensors.

Self-describing data transfer requires information about the meaning of the data to be included as part of the transfer [5]. These *meta-data* must include all information needed to interpret the actual data stream. For example, the time-invariant properties of a remote sensor that might be relevant include the sensor's location, the units of measurement, and the precision of its measurements. In addition, a description of the algorithm used to extract the desired information from the data is required.

Any successful methodology that provides self-describing data transfers must meet the following criteria:

1. The transfer includes all information *meta-data* needed to interpret the data, together with the data themselves. If this requirement is met, the data transfer is *self-describing*.

2. The transfer method can be applied to a broad category of data types and procurement methods. This is a requirement for *data type independence* of the data transfer method.
3. The transfer method is applicable to a wide variety of computing environments. This is a requirement for *portability and general applicability* of the data transfer method.

Strictly, only the first requirement need be met to qualify the data transfer as self-describing; the effect of the other requirements is to enhance the generality of a transfer method. A variety of proposed data transfer mechanisms transfer the data and meaning [6, 7, 8, 9, 2, 3]. Many of the available data sharing methods involve the construction of custom software that "understands" the meaning of the data to be transferred for each class of data transfer [3, 2]. Such methods fail the second and third criteria listed above. They fail the second criterion because the transfer is specific to one type of data. They fail the third criterion unless the custom software is written in a portable manner.

We present a new approach to solving the self-describing data (SDD) transfer problem. Our data transfer method serializes a data description, in the form of a Data Dictionary, with the actual data to be transferred. Our data description makes use of the power of database query languages to ease the task of constructing a *Data Dictionary* to contain the necessary meta-data. Database languages are well suited for the task at hand because they are designed for the description and categorization of data. This Data Dictionary is the initial part of an SDD transfer, and the actual sensor data are serialized after the Data Dictionary, as shown in Figure 1.1. An SDD transfer is composed of one Data Dictionary and a continuous stream of sensor data. An SDD transfer ends when a new Data Dictionary is transferred. We are proposing the SDD transfer method presented here as a robust mechanism for distributing ITS data to Information Service Providers[1] (ISPs). We are aware of two other efforts to produce standards for communication of

[1]See the ITS National Architecture study for more information on ISPs [10, 11]

transportation related data. They are the National Transportation Communications for ITS Protocol and the Transportation Network Profile of the Spatial Data Transfer Standard. These standards were designed to meet significantly different needs, though both are designed to solve the problem of standardized data communication. The SDD paradigm presented here is related to aspects of both standards.

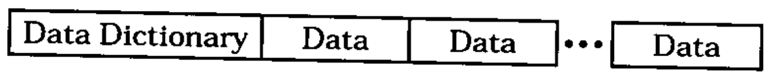


Figure 1.1: Data model for self-describing data transfers

The first related standard is the National Transportation Communications for ITS Protocol (NTCIP). NTCIP is a family of communications protocols (A, B, C, E) developed for real-time communication between a master controller and field devices such as traffic signal controllers, environmental sensor stations, dynamic message signs, highway advisory radio, closed circuit TV, and freeway ramp meters [12]. For example, the class B protocol is designed for direct communication between a master controller and one or more field devices connected by a communications cable for low-speed data transmission (on the order of 1200 baud). The target application is the control of traffic signal management systems. The standard covers both the rules of transmission and the format and meaning of a set of standardized messages to be transmitted.

NTCIP is basically an extension of the Simple Network Management Protocol (SNMP) [13]. It extends the management information base hierarchical name-space defined with Abstract Syntax Notation (ASN.1) [14] for use with SNMP. Groups of objects defined in this tree structured name-space are referred to as Management Information Bases (MIBs) and represent the set of objects (in the object oriented design sense) that are needed to effect the desired control or information transfer operations. NTCIP is intended for many types of transportation devices, each with different database requirements. Control or data exchange is effected by modification of the objects in the MIB associated with the device(s) being controlled. This modification uses the get/set paradigm of the SNMP to change the values of objects in the MIB. The resulting action depends on the programming of the controlled devices.

NTCIP is designed to standardize the control of traffic management devices such as traffic light controllers. The method uses a Data Dictionary, the management information base (MIB), as a repository of information to control the external devices. A device's response to some change to its MIB is determined by software resident at the device. Under that paradigm, to request data from a sensor, a management application modifies the sensor's MIB in a way that allows the management application to obtain the sensor data.

NTCIP is a control protocol that uses a globally agreed upon set of objects. To be effective, the control/management software that uses the get/set operations must understand how the software within the devices reacts to changes in the MIB. So an a priori agreement on software functions, as well as on object definitions, is necessary.

From an NTCIP perspective, SDD transfers can be cast as a compound object for information transfer without a priori information about the data to be shared. Thus, SDD is targeted at information transfer and not control. The control function of NTCIP can be used to initiate or terminate SDD transfers. The Data Dictionary components and the actual data can be declared as objects, and an ASN.1 compound object can be created. Changes in the MIB structure will then initiate or terminate an SDD transfer by using the SNMP paradigms. The SDD transfer can be implemented either by viewing the MIB as a control that initiates an out of band data transfer, as in [15], or the MIB can actually contain the components of SDD as objects. SDD transfers leverage NTCIP in that SDD has an agreed-upon data-description language that includes methods to describe and extract the elements of data from a data stream without the need for a great deal of a priori knowledge. For example, it is possible to create an application that would operate in a Java environment and that could obtain the methods from the Data Dictionary in the form of Java language elements. Those methods could operate directly on the data stream, creating an automated transfer of data with only SQL and Java as the required a priori knowledge.

The second related standard is the Transportation Network Profile (TNP) of the Spatial Data Transfer Standard (FIPS 173) that is designed for use with geographic vector data that have network topology. The TNP allows transfers of spatial data that can be represented by vector objects that make up a network or planar graph. The TNP data types are *nodes* and *links* between nodes, each of which may have associated attributes. These associated attributes may be multi-valued and, therefore, could be used to convey time-varying information associated with a node or a link.

The TNP provides a mechanism for defining an external Data Dictionary module that need not be included in each transfer, thereby reducing the amount of overhead that must be devoted to sending a dictionary module for multiple transfers that use the same dictionary.

Though it was not designed specifically for the purpose of transferring large amounts of time-varying data, the SDTS/TNP could be used to transfer time-varying data from a set of remote sensors. The time-invariant information about the sensors (location, sensor type, etc.) could be represented as single valued attributes of the nodes that make up the network, and the time-varying data would be represented as multi-valued attributes functionally dependent on time. If the time dependent attributes were isolated into a separate table from the time invariant information, it would be possible to send the module containing time-invariant data first and then continue with the "open-ended" time-varying module until no more time-varying data were desired at the receiving end. Though possible, transfer of large amounts of time-varying data with the SDTS/TNP has two disadvantages: (1) the data stream must be interpreted before transfer and placed into the appropriate attribute tables, possibly at a significant cost in required bandwidth, and (2) the overall design of SDTS/TNP does not really include the notion of a transfer of indeterminate length. Our method makes a clear distinction between time-invariant data (the Data Dictionary contents) and time-varying data (the actual data stream) and allows for a more efficient treatment of the actual data stream.

1.2 Data Model

In the work presented here, the data to be transferred are modeled in two components: (1) the Data Dictionary and (2) the actual sensor data. These two components, transferred serially, effect an SDD transfer. The Data Dictionary component is central to our self-describing data transfer method. In our data model, the Data Dictionary comprises two parts: (1) Dictionary Schema and (2) Dictionary Contents. These two parts provide the necessary description of the data to make them useful to a client and are described in the next sections.

1.2.1 Dictionary Schema

The first part of our Data Dictionary is the *Dictionary Schema*. This comprises meta data that specify the schema of the data description (e.g., a sensor has a name, position, and units of measure, and the position is specified in latitude and longitude). The Dictionary Schema is a provider-defined database schema written in a subset of Entry Level SQL-92 [16]. We chose Entry Level SQL-92 because it is fully relational, and the relational model can represent an arbitrary set of data [17]. The SDD model presented here allows for the construction of any schema that SQL allows, and this guarantees a powerful data-description language. In the Dictionary Schema, the data provider should include sufficient information about the actual data to allow a recipient to interpret those data. Because the sufficiency of the Data Dictionary is dependent on its author, it is clear that the Data Dictionary concept *allows* for self-describing data transfer, but it does not *ensure* that any given data transfer is in fact self-describing. It would seem difficult, if not impossible, to make such an assurance in an automated system. It is, however, possible to automate verification that the schema provided conforms to the data description language.

As part of the SDD transfer method, we have created a Schema Parser which is used to verify the Data Dictionary schema definition. The language accepted by the parser is a subset of entry level SQL-92 that allows definition of schemas, tables, etc. but does not include any of the query processing facilities of a complete database language. Our intent in defining the Schema

Language is to provide sufficient power for the definition of a Data Dictionary while simultaneously making the language simple enough that it is easy to learn.

The Schema Parser is used in two ways by our self-describing data transfer protocol. First, the sending application uses the parser to verify that a user-provided schema definition is valid. The second task of the parser is the construction of a parse tree that is subsequently used to verify that the Dictionary Contents are compatible with the defined Dictionary Schema. The receiving application uses the parser again to verify that the received Dictionary Schema is a valid one and constructs a parse tree that helps to create and verify the Dictionary Contents file.

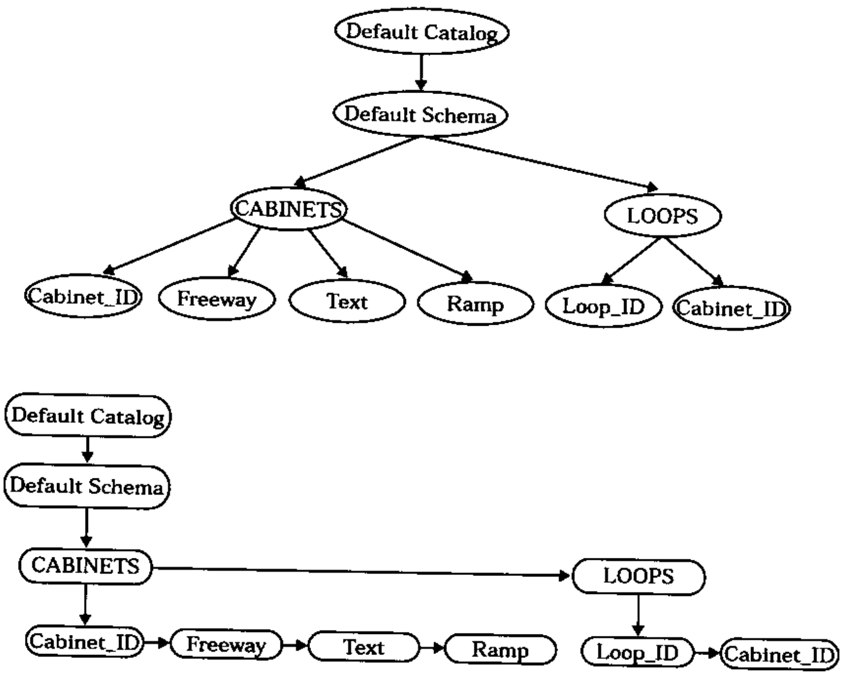


Figure 1.2: Parse tree structure at top and its leftmost child; right sibling representation at bottom

The parse tree is a memory-resident data structure that is constructed from the schema definition. The tree is implemented as a "leftmost child, right sibling" data structure in which the descendants of a node are represented as a linked list of nodes ordered from leftmost child to rightmost child. A node has two associated linked lists: a siblings list and a descendants list. Figure 1.2 shows an example parse tree and the list structure used to instantiate that tree.

The parse tree is used during schema verification to facilitate certain necessary. For example, no two tables may have the same name within the same schema, and no two columns within the same table may have the same name. When a name is encountered, it is inserted into the parse tree and checked for uniqueness at that time. An additional check is performed on the name to ensure that the columns named in a foreign key reference are compatible between the referencing and referenced tables. The parse tree contains sufficient information to make such checks and is organized so that the checks can be performed efficiently.

The structure of the Schema Language is such that the maximum depth of the parse tree is four levels. Furthermore, at each level, the nodes are of a specific type. The four types, in order of increasing depth in the parse tree, are catalog, schema, table, and column. For entry level SQL-92, the catalog and schema nodes are not really needed since only one of each can exist. Asingle schema node could be used as the root of the parse tree. We chose the four-level tree implementation to make it easier to extend to higher levels of SQL-92 conformance (which allows multiple schemas and catalogs).

1.2.2 Dictionary Contents

The second part of the Data Dictionary is the *Dictionary Contents.* The Dictionary Contents comprise the information about the actual data stream that can be used to construct a database describing the static information about the sensors for this data transfer. This is the component that contains particular values for the description of each sensor (e.g., sensor one is at 122.23 deg longitude, 47.21 deg latitude, and measures rainfall in inches). We define a *Contents Language* and associated parser to facilitate verification that the schema contents are compatible with the schema into which they will be placed. A Contents Parser recognizes the language used to describe the contents of the Data Dictionary. The language is designed to allow specification of the table and columns into which a set of data tuples will be inserted.

The contents language is fairly simple. A *data dictionary file* consists of a series of *table entries*. Each table entry is of the following form:

```
TABLE <table name>
COLUMN (<column name 1>, ... , <column name n>)
<data for column name 1>, ... , <data for column name n> ;
... one tuple for each row to be inserted into the table
<data for column name 1>, ... , <data for column name n> ;
```

The parser ensures this format and ensures that the type of data supplied in each tuple matches the data type declared for its corresponding column.

An abbreviated version of the contents file used with the transmitter described in Chapter 2 is shown in Appendix F . Its associated schema file is shown in Appendix E.

On the sending end of a data transfer, the Contents Parser helps to verify that the contents file supplied by a data provider is compatible with the Dictionary Schema in use. On the receiving end, the Contents Parser helps to verify that the data in the contents file are compatible with the schema during SQL command generation.

1.2.3 Data Transfer

The overall architecture of our system for self-describing data transfer is shown in Figure 1.3. This structure is independent of the actual data stream involved. The data transfer is a serialized stream divided into frames. The first frame of a transfer contains the Dictionary Schema. The second frame contains the Dictionary Contents. Subsequent frames contain the most recently available set of data from the data source. The implementation of a self-describing data transfer takes the form of a transmitter and a receiver.

1.2.4 Transmitter

Figure 1.4 shows the components necessary to construct and transmit a stream of self-describing data. The SDD transmitter verifies the compliance of the schema using the Schema Parser, which creates a parse tree as a byproduct of the compliance check. The Contents Parser then uses this parse tree to verify that the contents comply with the schema. If both the schema and

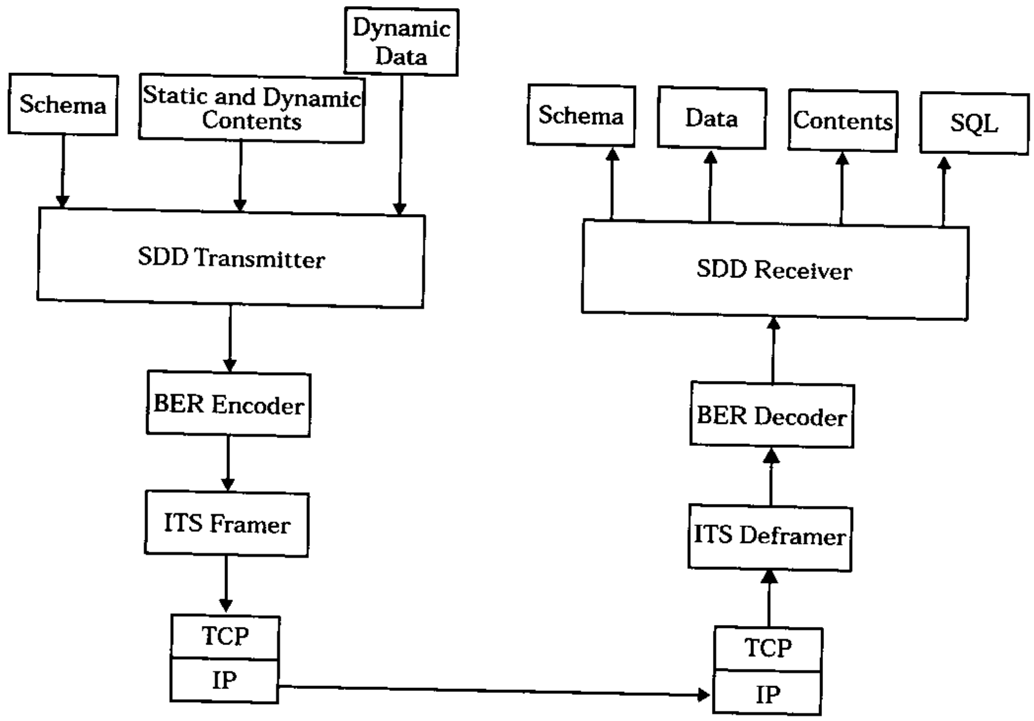


Figure 1.3: An overview of the structure of our system for self-describing data transfers.

the contents are in compliance, they are serialized in the following order: (1) schema, (2) contents, and (3) the actual sensor data. Transfer of the dictionary and data between computer systems is accomplished by encoding the data according to the Basic Encoding Rules (BER) defined for the ASN.1 standard. As a block of data is received, it is encoded and sent to all clients via the redistributor methodology detailed in [15]. In our application, the data are a stream of bytes, and the information is extracted with algorithms specified in the Data Dictionary. This mechanism allows for a very general transfer that is easy to encode but that requires the transfer recipient to devote programming effort to data extraction.

BER encoding of the self-describing data stream involves three types: data, schema, and contents. Each of these types is encoded according to the BER standard (ISO 8825-1) [18] with a type, length, and value.

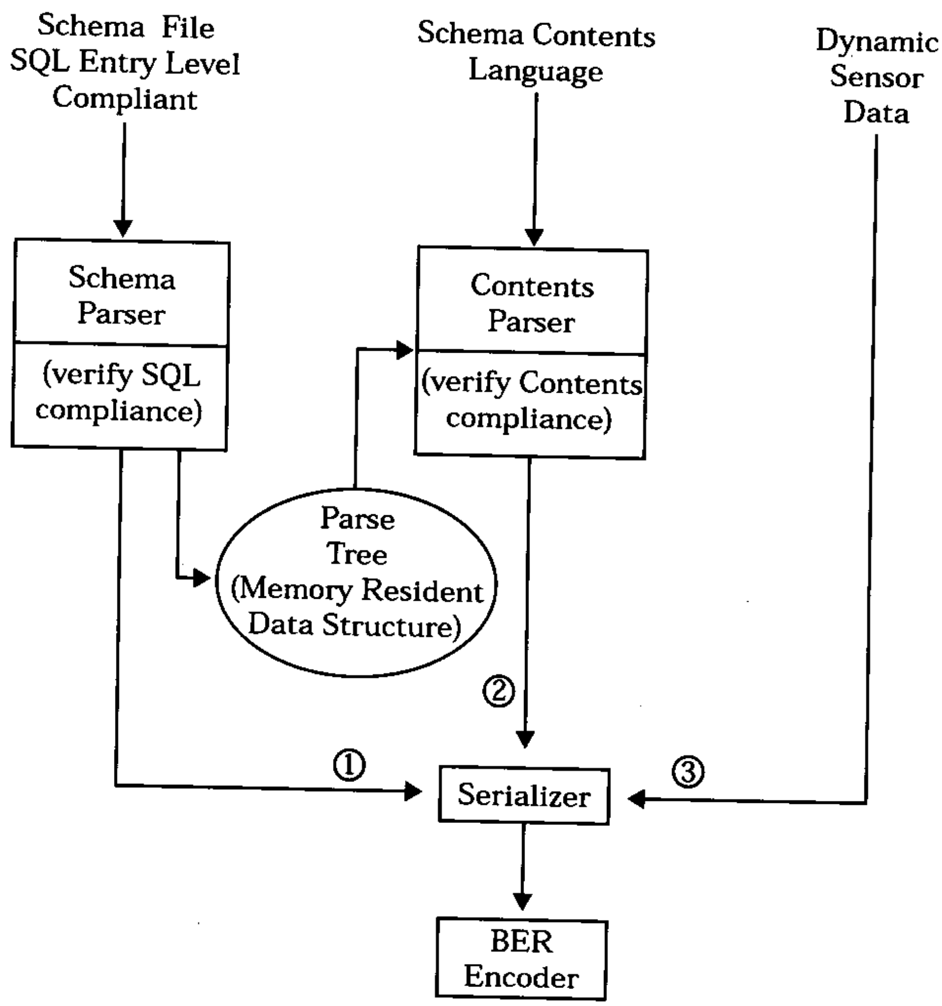


Figure 1.4: Self-describing data transmitter

We use the ASN.1 "Application" class with the "primitive" encoding. Tag numbers 1, 2, and 3 denote schema, contents, and data, respectively. Our identifiers are therefore encodable in one octet. We encode the "schema" and "contents" types as IA5 strings, and the "data" type is unchanged during encoding. An ASN.1 type declaration of our types is as follows:

```
DictionarySchema ::= [APPLICATION 1] IA5String
DictionaryContents ::= [APPLICATION 2] IA5String
Data ::= [APPLICATION 3] OCTET STRING
```

The tag number must be unique across the family of protocols using these three data types. For example, if this system was used with the NTCIP family of protocols, the NTCIP standards document would need to dedicate three types to an SDD transfer facility.

The serializer in Figure 1.4 sends a type, length, and value to the BER Encoder. The BER Encoder encodes these values as the appropriate header, length bytes, and contents bytes according to the ASN.1 definition above.

1.2.5 SDD Receiver

The SDD Receiver is a client application that converts data from the data transfer stream format back to three data sets: schema, contents, and data. The structure of the receiver, as shown in Figure 1.5, is parallel to that of the transmitter in Figure 1.4. The BER Decoder takes an SDD data stream as an input, and BER decodes the data stream and provides a decoded serial data stream that has the structure described in Figure 1.6. The bottom portion of Figure 1.6 is the structure for each of the frames (Scheme, Contents, and Data) in the top part of the figure. Each frame contains both a BER encoded serial number and the data appropriate for the frame. The serial number is a 17-character timestamp with the format 'yyyymmddhhmmssmmm' (a four-digit year designation, followed by a two-digit month, a two-digit date, a two-digit hour, a two-digit minute, a two-digit second, and a three-digit millisecond). The serial number is used to maintain state and to relate meta-data to data. The specific relationships between the serial numbers and the individual data types are that (1) Contents and Data must have the same serial numbers and (2) the Contents/Data serial numbers must be greater-than-or-equal-to the Schema serial numbers. The second BER packet contains the SDD data type packet, which contains either meta-data (Schema, Contents, and Extractor) or binary data. The BER Type Demultiplexor receives a serial data stream from the BER Decoder, and it uses the BER type field to distribute the schema and contents to the appropriate parser. The parser components of the receiver are identical to those of the transmitter. The schema component is sent to a Schema Parser that verifies SQL-92 compliance and creates a parse tree for use by the Contents Parser. The contents are sent to the Contents Parser, which

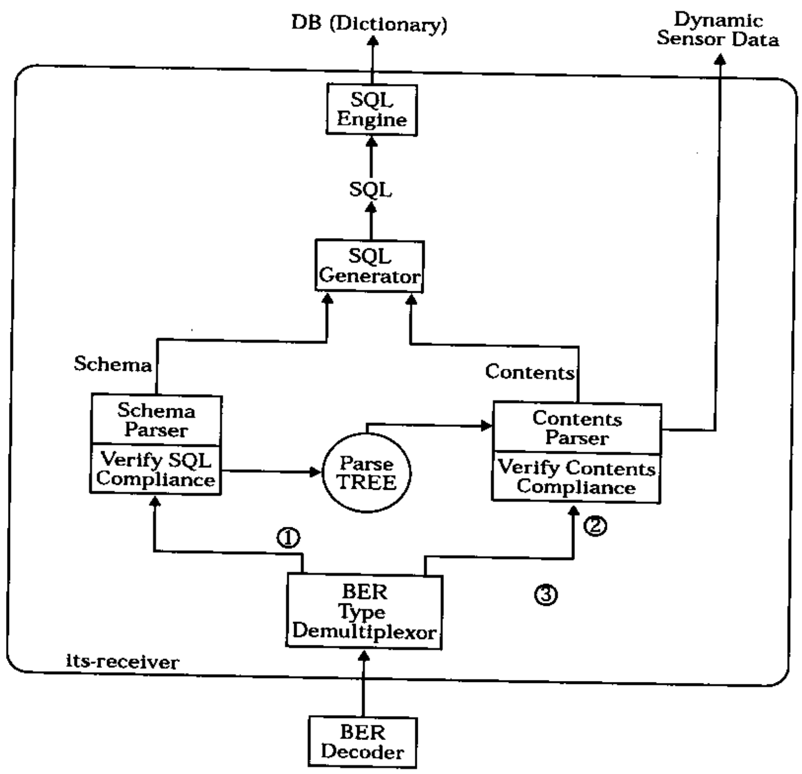


Figure 1.5: The detailed structure of the SDD Receiver application

verifies the contents against the schema. The outputs of these two parsers are the verified schema and contents used to describe the dynamic sensor data. With the arrival of the data dictionary schema, data dictionary contents, and the sensor data, a self-describing data transfer is complete.

The practicality of this paradigm is demonstrated in the next section, which uses an ITS example.

1.3 ITS Example

The Washington State Department of Transportation (WSDOT) operates a Traffic Management System (TMS) to collect data from traffic sensors in the central Puget Sound region, and these TMS traffic data are the focus for this ITS application of SDD. The data from the TMS consist of three parts: (1) sensor data, which represent measures of traffic conditions (e.g., speed, flow, occupancy), (2) information about the location and type of each sensor (e.g., mile marker, latitude, longitude, calibration), and (3) lists of presently available sensors. The sensor data come

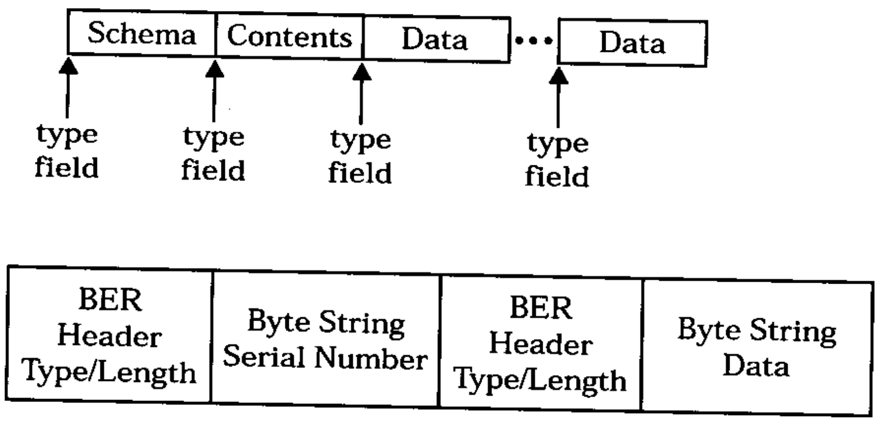


Figure 1.6: Structure of our serialized data stream for self-describing data transfers

from inductance loop detectors placed at approximately 3000 locations in the Seattle metropolitan area, and the list of sensors/detectors presently available changes slowly with time (e.g., every few hours). This dynamic list of available sensors makes up the dynamic component of the data dictionary contents. The name, location, measurement type, and other information make up the static component of the data

Figure 1.7 provides details about our "tms2sdd" application, which converts legacy TMS data to our self-describing transfer format. The application can be divided into a "legacy" component and a "standard" component. The legacy component is dependent on the specific data source; the standard component does not change from one source to another. As in the generic transmitter case, the self-describing Dictionary Contents file is verified against the Dictionary Schema by the standard component of tms2sdd. The schema is verified by the Schema Parser, which constructs a memory-resident parse tree that is used by the Contents Parser during the process of parsing and verifying the Dictionary Contents file. If verification of both files succeeds, they are transmitted to the BER Encoder.

Operationally, when the TMS application is started, a block of meta-data is retrieved from the TMS as an ITS Frame (as shown at the top of Figure 1.7). The meta-data block received from the TMS Data Stream represents that part of the Dictionary Contents that is subject to relatively frequent change. To construct a complete Dictionary Contents file, the tms2ddc function combines

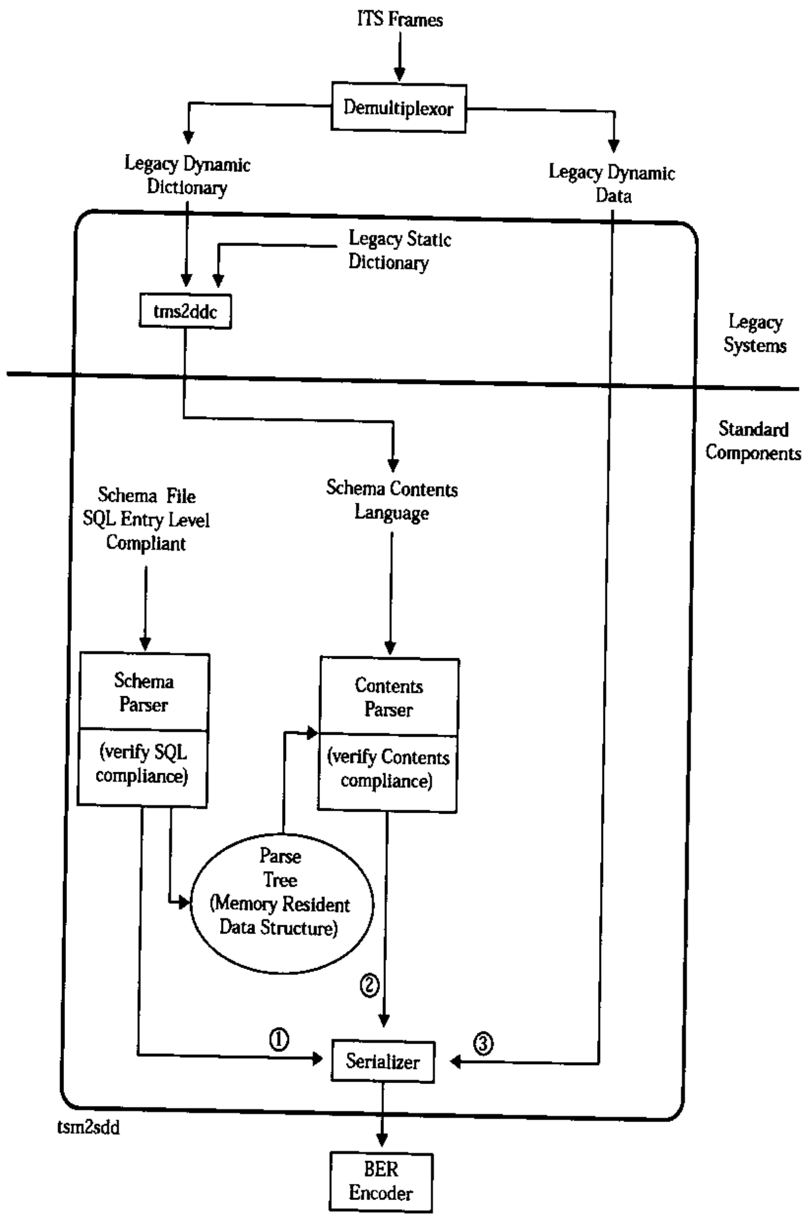


Figure 1.7: The structure of our example application for self-describing data transfers.dictionary contents.

the contents of a Static Data File(s) with the information in the meta-data block to construct a complete Dictionary Contents File. The Data Dictionary is then transmitted in two parts: the Dictionary Schema and the Dictionary Contents.

In this example, the Data Dictionary embodies the notion of state, in that the sensor data stream has an unambiguous interpretation once the Data Dictionary is present and is otherwise meaningless. Whenever a block of meta-data is received from the TMS Data Source indicating that a change in the number, type, or availability of loop detectors has taken place, a new Data Dictionary is created. The transmission of this new Data Dictionary signals the end of one SDD transfer and the beginning of another (or in other words, a change of state). Transfer of the Data Dictionary occurs when a client first connects and subsequently whenever a block of meta-data is received in the TMS data stream. Once the Data Dictionary has been sent to a client, the actual data are transmitted as they become available (in our application, a new block of sensor data arrives every twenty seconds).

In our implementation, we have added an SQL generator, as shown at the top of Figure 1.5. The SQL Generator creates a series of SQL INSERT commands from the Dictionary Schema and Dictionary Contents. Each data tuple in the contents file will be represented by an INSERT statement in the SQL output file. The commands, when used as input to an SQL database engine, will create and fill a database that instantiates the Data Dictionary. We include this step as a practical matter for the engineering users of the SDD paradigm, so that the results of an SDD transfer are a data base with which an engineer can interact and a binary stream of dynamic sensor data. The SDD methodology described here is being deployed for ITS traveler information applications in the Seattle metropolitan area. Information service providers can connect to the regional ITS backbone using one receiver and obtain a variety of data types.

2. Self-Describing Data Transmitter Implementation

This chapter explains in detail the implementation of the self-describing data transfer system discussed in Chapter 1, Section 1.3. The intent of this chapter is to provide sufficient detail for an experienced programmer to construct an implementation of the self-describing data protocol. Our implementation uses a *Generic Redistributor* developed by the UW ITS Research group to handle transport over a network and client connection details. We have modified our Generic Redistributor to implement the self-describing data transfer protocol as a data source. Though we have used our Generic Redistributor to implement the self-describing data tranfer protocol, the Generic Redistributor is not itself an integral part of that protocol.

The following discussion includes references to specific functions defined in the implementaion.

2.1 Generic Redistributor

The Generic Redistributor is an application that receives input from a single source and redistributes that input to a set of clients. It consists of three processes:

1. an Input Buffer,
2. a Connection Daemon, and
3. an Output Multiplexor.

The Input Buffer manages incoming communications. The Connection Daemon manages connection requests from clients. The Output Multiplexor manages redistribution to the set of clients. It may cache certain items (such as the legacy Data Dictionary) to be transmitted to newly connected clients.

The data units handled by the Generic Redistributor are *ITS frames*, which consist of a type header and a length value, followed by a number of contents bytes equal to the length value. The redistributor uses the type associated with an ITS frame to determine what to do with an incoming frame. The types DATA and DICTIONARY are of particular concern to this discussion; the

Generic Redistributor caches the most recently received DICTIONARY frame and transmits it to any newly connected client and simply passes DATA frames on to the current set of clients without caching them.

2.2 SDD Transmitter

The SDD transmitter converts a legacy data stream into a self-describing data stream and distributes it over a network to a list of clients. We based our implementation on the Generic Redistributor described in Section 2.1. This section focuses on the changes that were made to the Generic Redistributor to convert it into an SDD transmitter.

The functions of an SDD transmitter can be divided into two logical sections: (1) the core SDD section and (2) the legacy to SDD conversion section. The core SDD section includes functions that must be part of any SDD transmitter, while the legacy to SDD conversion section includes functions that must be modified to suit a specific data source.

The SDD transmitter described here was constructed by modifying the Generic Redistributor described in Section 2.1. The required modifications alter the Output Multiplexor element of the Generic Redistributor in two ways:

1. An incoming legacy dictionary is handled by a function that is responsible (1) for converting the incoming legacy dictionary (contained in an ITS frame of type ITS_T_DICTIONARY) into an outgoing schema and contents and (2) for packaging the schema and contents into BER encoded frames. The Generic Redistributor then packs the BER encoded schema and contents into ITS frames of type ITS_T_SCHEMA and ITS_T_CONTENTS. The schema and contents frames are cached and sent to any newly connected client before any data are sent to that client.
2. The contents of an incoming data frame (an ITS frame of type ITS_T_DATA) are BER encoded and then re-packed into an ITS frame of type ITS_T_DATA and finally distributed to clients by the Output Multiplexor.

Conversion from legacy dictionary to SDD dictionary involves elements of both the core SDD and legacy to SDD sections mentioned above and described in detail below.

2.2.1 Core SDD

The core SDD consists of a set of (1) parsers that recognize the schema and contents languages and (2) BER encoding routines that encode schema, contents, and data. The properties of each of these are discussed in the following sections.

2.2.1.1 Schema Parser

The Schema Parser can be used by either a transmitter or receiver of self-describing data. The functions performed by the Schema Parser include the following:

1. verification that the input schema is part of the SDD Schema Language
2. verification that the semantic constraints of the Schema Language are met
3. construction of a parse tree for use internally by the Schema Parser and externally by the Contents Parser during verification of the schema and contents.

The *SDD Schema Language* accepted by the Schema Parser is a subset of entry level SQL-92 that allows creation of schemas and tables but lacks query capabilities and other aspects of the complete SQL language. The subset is demonstrably sufficient to construct any legal SQL relation schema and to generate any relation on that schema. The yacc (a LALR parser) grammar definition for the Schema Language is found in Appendix A, and the flex scanner definition is found in Appendix B.

A parse tree is constructed by the schema parser during processing of the schema file. An illustration of an example parse tree is shown in Figure 1.2. The parse tree has a maximum depth of four levels because of the nature of the Schema Language. At each level of the tree, all nodes are of one type. The root node is a catalog node, the children of the catalog node are schema nodes, the children of schema nodes are table nodes, and the children of table nodes are column nodes. Since column nodes can have no descendants, the maximum depth of the tree is limited to four levels. Each node type provides auxiliary storage for information about the specific entity represented by the node. Because that information is different for each node type, the auxiliary data structures are node type specific.

Construction of the parse tree is controlled by the Schema Parser. Appropriate calls to tree construction routines are placed in the yacc specification so that tree nodes are added or updated when the needed information has been recognized by the parser. The actions taken when each of the four node types is recognized are described below. Because the data structures used are somewhat complex, the type definitions are included here to aid in the discussion. The basic parse tree node type is as follows:

```
typedef struct symbol_node {   /* generic tree node */
  char ID[MAX_NAME_LENGTH];
  NodeClass class;      /* enumerated, one of CAT, SCH, TAB, COL */
  NodeInfoType node_data;
  symbol_node_t left_Child;
  symbol_node_t right_Sibling;
} symbol_node;
```

The structure includes fields for a name, a node class (one of CAT, SCH, TAB, or COL), an auxiliary structure to contain information relevant to the particular node class, and list headers for children and siblings that are used to implement the leftmost-child, right-sibling tree structure that represents the parse tree. NodeInfoType is a union of structure types catalogInformation, schemaInformaion, tableInformation and ColumnInfo. The structures are customized to contain the data relevant to each of the possible node types.

2.2.1.2 Parse Tree Structure

The parse tree is a general tree implemented with a leftmost-child, right-sibling linked list data structure. Each node has pointers to two other tree nodes. One points to the leftmost child of the node, the other points to the node's right sibling. Given this representation, the descendants of a node are found by following first the leftmost-child pointer and then the right-sibling pointer of that child until no more siblings of the child are found. This data structure allows an arbitrary number of children of any node but only requires space proportional to the actual number of children of the node.

A set of tree construction and traversal routines are defined to facilitate construction and searching of the parse tree. They include Create_root(), Find_node(), Insert_node(), Delete_node(), Update_node(), Preorder(), Add_unresolved_node(), Check_unresolved_node(), and ClearUseFlags(). These routines allow for insertion to, deletion from, and searching and updating of the parse tree and are used during construction of the parse tree and verification of required semantics.

For example, a foreign key (a list of columns in a different table that represent a key to that table) can be defined before all columns involved (or even the foreign table itself) have been defined. A check of unresolved names is performed after the parse has completed to verify that this semantic requirement is met.

2.2.1.3 Parse Tree Node Information Structures

In this section, the information structures used with each of the parse tree node types are described.

The catalogInformation structure used with a Catalog node is defined as follows:

```
typedef struct cat_info {
  char attribute[16];
} catalogInformation;
```

Entry level SQL-92 does not allow user-specified catalogs. A default catalog is placed into the parse tree before any other parsing actions take place. The default catalog node has a name and no other attributes. Expansion of the Schema Language to include higher level compliance to SQL-92 must allow for multiple catalogs. Presently, no information is stored in the catalogInformation structure.

The schemaInformation structure used with a Schema node is defined as follows:

```
typedef struct schema_info {
  int count;
  char attribute;
} schemaInformation;
```

Entry level SQL-92 does not allow user-specified schema names. A default schema node is inserted into the parse tree before parsing begins, immediately after creation of the default catalog node. Expansion of the Schema Language to higher-level compliance with SQL-92 must allow for user-specified schemas. Presently, no information is stored in the schemaInformation structure.

The tableInformation structure definition is

```
typedef struct table_info {
  queue_t primary_key; /* list of columns that constitute the primary key
*/
  queue_t foreign_key; /* list of ForeignKeyInfo records */
  queue_t unique; /* list of lists of columns in UNIQUE constraints */
} tableInformation;
```

A new table node is inserted into the parse tree when a table name has been recognized. Additional attributes to a table node include a list of columns that constitute the table's primary key, a list of foreign keys defined in the table, and a list of unique specifications for the table. Each of these is added to the tableInformation structure as the information becomes available during parsing.

Only one primary key can be defined for a table. It is represented as a list of column names.

A table may contain several foreign key definitions. They are represented by a list of records of type ForeignKeyInfo, defined as follows:

```
typedef struct f_key_info {
  queue_t local_cols;
  queue_t foreign_cols;
} ForeignKeyInfo;
```

The columns local to the table are contained in the list "local_cols," and their definitions must match those of the corresponding columns in the list "foreign_cols." Verification of that semantic requirement is performed by the function *CheckTableConstraints()*.

A table may contain several "UNIQUE" constraints. A unique constraint requires that a tuple of entries into the constrained columns is unique within the table. Columns referenced by a foreign key in a different table must be either declared as a unique constraint or as a primary key. Each unique constraint is represented by a list of column names, and the UNIQUE list is a list of unique constraints. The function *CheckTableConstraints()* verifies that the foreign columns of any foreign key definition are either a primary key or are constrained to be unique.

The ColumnInfo type is defined as follows:

```
typedef struct column_information {
  DataTypes datatype;
  data_type_info dataTypeInfo;
  char defaults[5];
  char constraint[32];
  int offset;          /* not used */
  int primary_key;          /* 1 if column is primary key, 0 otherwise */
  int InUse;           /* flag to avoid duplicate columns in contents lang */
} columnInfo;
```

A new column node is added to the parse tree when a column definition is recognized during parsing. At that time, uniqueness of the name within its table is checked. Additional information about the column includes a data type specification, a default value, and possible column constraints.

The information needed for a column's data type specification is dependent on the actual data type. The dataTypeInfo field is a union type that allows the information to be customized to the specific data type of the column. The relevant type declarations are shown below. The types are assembled into a union of structures specific to the basic data types described below:

```
typedef union d_type_info {          /* basic datatypes */
  StringTypeInfo StringType;
  numericTypeInfo numericType;
  datetimeTypeInfo datetimeType;
  intervalTypeInfo intervalType;
} data_type_info;
```

String types are represented by the StringTypeInfo structure:

```
typedef struct str_type {
  int ConstantLength;   /* 0 for variable length, 1 for constant length */
  int length;           /* the max length of the string */
} StringTypeInfo;
```

Entry level SQL-92 does not allow variable length strings, so the ConstantLength field is not really needed. It is included to ease the task of expanding the Schema Language to higher levels of compliance.

Numeric types are represented by the numericTypeInfo structure:

```
typedef struct numeric_type {
  NumericTypes datatype;
  int precision;     /* precision >= scale */
  int scale;         /* default scale is 0 */
} numericTypeInfo;
```

The allowable types, specified in the enumerated type NumericTypes are numeric, decimal, integer, smallint, float, real, and double precision. The scale and precision fields are relevant only to types numeric, decimal, float, real, and double precision. The precision for real and double precision is implementation defined; the precision for numeric, decimal, and float may be specified by the user.

Datetime and interval types are not allowed in entry level SQL-92, but the definitions are included to ease the task of expanding the Schema Language to higher compliance levels:

```
typedef struct datetime_type {
  DTTypes datatype;
  int precision;
  char timezone[16];
} datetimeTypeInfo;

typedef struct interval_type {
  IntervalFieldTypes startfield;
  int startprecision;
  IntervalFieldTypes endfield;
  int end_ld_prec;
  int end_sec_prec;
} intervalTypeInfo;
```

In addition to type information, a column could be the sole element of the primary key for a table. If so, that fact is indicated by setting the primary_key flag to TRUE. The information is also placed in the primary_key list for the table containing the column.

The only allowable SDD Schema Language default value is NULL, and if that is defined to be the default value for the column, the defaults field is set to contain the string "NULL."

Allowable column constraints are stored in the constraint field. Allowable constraints include NOT NULL and UNIQUE. If a column is declared UNIQUE, it is added to the list of UNIQUE column groupings for the table of which it is a part as a single-column UNIQUE specification.

2.2.1.4 Semantic Checks

When the end of the input schema is reached, the parser performs semantic checks before returning. The first semantic check verifies that all names encountered during parsing have been defined. The check is performed by the routine *Check_unresolved_node()*. During parsing, a list of undefined names is maintained, and if it is non-empty at the end of parsing, its contents are checked by searching the parse tree for each previously undefined name. If all names are found, all is well; otherwise, an error has been detected, and the parser returns with an indication of failure.

If the unresolved name check succeeds, then a check of table constraints is performed. The check verifies that any foreign key declarations are valid by verifying that the type defined for each of the corresponding referencing and referenced columns matches and that the foreign key is either a primary key in the referenced table or the set of columns has been declared UNIQUE within that table. These checks are performed by the function CheckTableConstraints(), which calls various auxilliary functions internal to the file, tree.c.

After the semantic checks have been performed, the parser returns either failure or a pointer to the parse tree that it has constructed.

2.2.1.5 Contents Parser

The purpose of the Contents Parser is to verify the syntax and semantics of the contents file by verifying that the tables and columns referenced in the contents file are defined in the schema and that the types of each data tuple present are compatible with the table and columns into which it is to be placed. The Contents Parser makes use of the parse tree constructed by the Schema Parser to perform the semantic verifications and to ensure that the list of columns into which a set of tuples will be inserted does not contain the same column name more than once (indicated by the "InUse" flag from the columnInfo structure).

The Contents Language is a simple language that allows specification of both a table name and a list of columns from that table into which a series of tuples will be inserted. The table must exist in the SDD schema, and the listed columns must exist in that table.

Each member of each of the tuples provided for insertion into a named table is checked for type compatibility with the destination column. The parse tree generated by the Schema Parser is used to aid this verification.

2.2.1.6 BER Encoding

Self-describing data are encoded with the Basic Encoding Rules (BER) [18] defined for the Abstract Syntax Notation One standard (ASN.1) [14]. We use this standard to help minimize machine dependency. We chose to hand craft a set of encoding routines because the encoding of our ASN.1 types is relatively straightforward and it would avoid requiring that an ASN.1 compiler be available.

We define four application specific ASN.1 types at present. They are schema, contents, data, and serial number. Soon to be added will be Java byte code (for the extractor application).

BER encoding is done by two routines: BEREncodeFileSN() and BEREncodeBufferSN(). These encode the contents of a file or memory buffer as one of the BER types utilized by the SDD protocol. The routines place the encoded input file or buffer into an output buffer and allocate space for that buffer if necessary. If the output buffer has already been allocated and is large enough, it is used; if not, it is reallocated with sufficient size. The arguments to each function are an input stream or buffer, an output buffer, the allocated size of the output buffer, the BER type to be encoded, a length in bytes of the buffer or file to be encoded, and a serial number. The functions encode first the serial number and then the file or buffer, placing both BER encodings into the same output buffer. The serial number is encoded as an IA5String, since it is a character representation of an integer that can be interpreted as a GMT date time with millisecond resolution. Encoding as an IA5String is done by the internal function IA5Encode(), which takes a character argument and encodes it as its IA5 representation. Currently, we use ascii and should revise the function to be in compliance with the IA5 standard. The BER length octets are constructed by the internal function EncodeLengthHeader(), which only deals with the definite form of the length octets as defined in the BER standard [18].

2.2.1.7 Client Connection Maintenance

To provide a self-describing data stream requires a client management system. That system handles connection to the source of data, requests for connection from clients, distribution of schema and contents to newly connected clients, and distribution of data to each client.

A method for managing client connection details is not specified as part of our self-describing data protocol. Our sample implementation makes use of the client connection management provided by the Generic Redistributor described in Section 2.1.

2.2.2 Legacy to SDD Conversion

The Legacy to SDD Conversion portion of an SDD transmitter converts information about the data stream (the legacy dictionary) into a format that is compatible with the self-describing data protocol. Its exact structure must depend on the conversion task that will be performed. The following description applies to the TMS loops data of our example.

2.2.2.1 Schema Creation

To convert the legacy TMS data into a self-describing data format requires construction of a schema for the available information about the data stream. That only needs to be done once (but revisions are required if there are changes to the data items available for a class of sensors). The schema should adhere to certain guidelines, as described below.

For each distinct data reporting element (sensor), a mapping between that unit's id (which may be a multiple component primary key if needed) and the location within a data block of the data reported by that element must be present in the schema. For easier construction of a data extractor, that mapping should be contained in a single table whenever practicable.

If data will be extracted from the data stream and stored in the database as they arrive, the schema should contain definitions of tables to contain the extracted data. See Appendix E for an example schema. When present, the tables will likely be used by a data extraction routine supplied in the algorithms table (and optionally as an executable Java application). Even if data are not inserted into the tables by an extraction routine, the data table definitions provide useful documentation of the information present in the data for each sensor type.

2.2.2.2 Contents File Creation

The legacy dictionary must be used to create a new version of the contents file whenever a new dictionary is received. The function CreateContentsFile() converts an incoming legacy Data Dictionary into a contents file. It makes use of several static files that contain information not present in the legacy Data Dictionary. This additional information located in flat files is needed for a complete description of the data stream. These files include a file containing a description of the algorithms needed to interpret the packed data stream format (theAlgorithmFile), a file containing information about the measurement systems used for position information (the MeasuresFile), and a file containing information about the loop sensors that is not present in the legacy Data Dictionary (the CabinetInformation file).

Because the routine that creates the contents file from the legacy dictionary and various static files depends on the schema definition, it must be changed if the schema definition changes.

A new contents file must be generated whenever an incoming dictionary differs from the one in current use. This is accomplished by the function labeled "tms2ddc" in Figure 2.1. Anincoming legacy dictionary might not be different from the one currently cached by the transmitter if it was sent by an upstream server after a re-start. Since contents file generation and transmission can require significant resources, we compare an incoming dictionary with the currently cached version to determine if it is a real revision and not merely a re-transmission from an upstream server and will only create a new contents file in the former case. This prevents constructing a new contents file and subsequent parsing and transmission of the schema and the contents to all connected clients if such activity is not necessary. Not only does this save transmission and processing costs, it also avoids requiring the recipient to generate a new database schema unless there is a real change to the Dictionary Contents.

2.2.3 Creating a Specific Transmitter

To create a transmitter application for a specific set of data requires the legacy portion of the transmitter to be modified. The steps required to customize a transmitter to a specific data stream are the subject of this section.

2.2.3.1 Define Schema

To be self-describing, a data set must include a Data Dictionary in the form of an SQL schema and the associated contents. The schema should include all available information about the data. To ease the receiver's task of interpreting the data, certain conventions should be followed. Two important conventions are the inclusion of a table in the schema that maps from sensor ID to the location of that sensor's actual data (the sensor-to-offset table) and the inclusion of an algorithm for interpreting the data, given the sensor's type and the location of its data.

The sensor-to-offset table may include other information but should at least include a unique key to each sensor and an offset into a block of data where that sensor's data can be found. In the present example implementation, data from each sensor are reported every 20 seconds as a binary stream. The offset into that block of data is given for each sensor by the legacy dictionary. The schema includes a table that holds the required mapping:

```
CREATE TABLE LOOPS
(LOOP_ID      CHAR(16)   NOT NULL PRIMARY KEY,
 CABINET_ID   CHAR(7)    NOT NULL,
 METERED      CHAR(4)    NOT NULL,
 RD_TYPE      CHAR(30)   NOT NULL,
 DIRECTION    CHAR(12)   NOT NULL,
 LANE_TYPE    CHAR(20)   NOT NULL,
 LANE_NUM     SMALLINT,
 SENSOR_TYPE  CHAR(12)   NOT NULL,
 DATA_OFFSET  INTEGER    NOT NULL)
```

In this situation, the LOOP_ID field is sufficient as a primary key. If, for example, a loop could provide information described by several different sensor types, then the SENSOR_TYPE field would be needed in the primary key as well. The DATA_OFFSET field contains a byte offset into a buffer of binary data from which a number of bytes that depend on the sensor type are processed by the extraction algorithm. The extraction algorithm must "know" the correct number of bytes to process for each sensor type.

The algorithms used to extract data into a usable form from the data stream should be documented in a table. Since entry level SQL-92 does not provide a "text" data type, our example application defines a table with 3 columns: line number, file name, and text:

```
CREATE TABLE ALG_DESCRIPT - Algorithms Used To Extract Data From
                          - The Real Data Stream.
(FILE_NAME CHAR(20) NOT NULL,
 LINE_NUMBER  INTEGER  NOT NULL,
 LINE_TEXT    CHAR(128),
 PRIMARY KEY (MEASURE_TYPE, LINE_NUMBER))
```

The contents of each file of the algorithm description can be extracted from this table by selecting LINE_NUMBER and LINE_TEXT for a given FILE_NAME, ordered by LINE_NUMBER. We recommend providing the algorithm description in the form of a Java program that performs the data extraction task. Since each Java class must be defined in its own file, the algorithms table separates information by file name. The algorithm description takes the form of textual Java code, and the recipient can compile the Java text to produce an executable data extractor. If the algorithm description takes some other form, then a custom data extraction routine must be constructed according to the specifications contained in the algorithms table.

2.2.3.2 Construct Contents

The contents file contains the information to be placed into the database according to the Dictionary Schema. In some instances the contents may not change with time or may change very slowly. In other instances the contents may change dynamically within a relatively short (hours to weeks) period. A dynamic legacy dictionary requires some automatic way to generate an update, while a static dictionary may only need to be defined once.

The TMS data of our example are described by a dynamic legacy dictionary. A dictionary can arrive in the data stream at any time. The TMS SDD transmitter constructs a new contents file from information contained in the legacy dictionary and several static data files whenever a new legacy dictionary arrives. To construct a new transmitter for a data transfer that has a dynamic dictionary, a routine that does the necessary legacy-to-SDD conversion must be created and added to the transmitter. In our TMS example, the function CreateContentsFile() performs the required conversion.

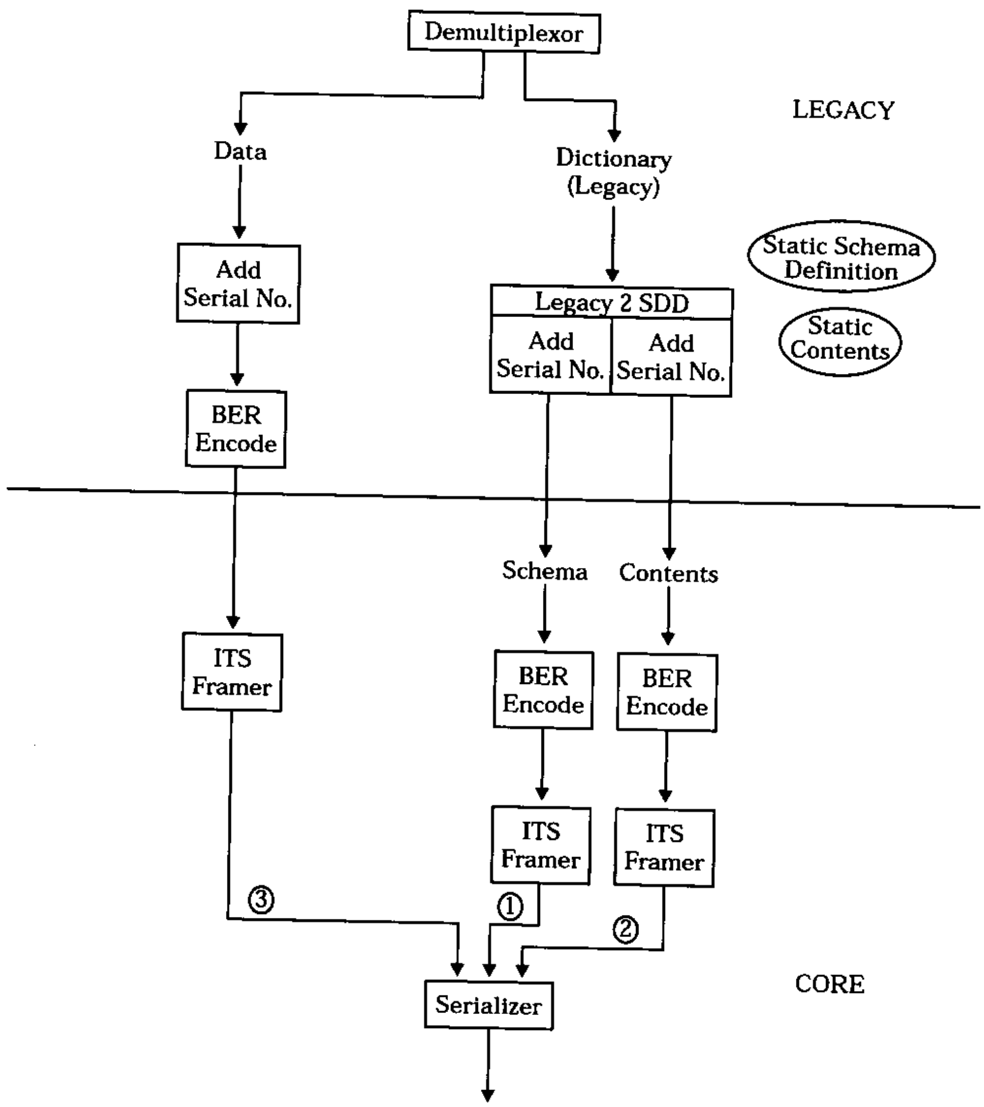


Figure 2.1: A new contents file is generated when an incoming dictionary frame is different than the one currently cached

If the legacy dictionary is static, the situation is considerably simplified. A contents file must be constructed before transfer can begin, but no provision for dynamic change to the contents file is necessary. The file can be generated by hand if it is small, or suitable tools can used to construct the file if it is large. Once constructed, the SDD transmitter can be modified to transmit the contents file when a new client is connected.

2.2.3.3 Construct Data Extractor

Self-describing data require the description to include information needed to interpret the data stream. Data streams are often transmitted in packed binary form for bandwidth reasons. If that is the case, then the information must include a description of how many data are present in the record for each sensor and the meaning of the individual bits in those data. An ideal way to convey such information is a program that accomplishes the required extraction. The wide availability of Java, together with its portability, make it well suited to the task of supplying a data extraction algorithm.

To construct a data extraction algorithm obviously requires knowledge of the sensor data format. If the algorithm description takes the form of Java code, it should be liberally commented to make that format clear to the reader of the code.

To obtain the mapping between sensor ID and offset, the extractor must have knowledge of the table structure of the schema. Specifically, it must know the name of the table that contains the required mapping. Since the extractor is constructed by the providers of the data, that knowledge should be available. The extractor could be constructed to parse the contents file and obtain the desired mapping from that source rather than performing a query to a database instantiation of the Data Dictionary.

If the data extractor is configured to generate SQL statements that place the extracted data into a database table, the extractor must have knowledge of the names of the tables and columns devoted to storing the sensor data.

2.3 SDD Receiver

Implementation of an SDD receiver makes use of many of the components of the SDD transmitter. It uses the same parsers for schema and contents as does the transmitter, and the BER encoding routines of the transmitter are the inverses of the BER decoders needed by the receiver. The receiver has the task of generating SQL commands that instantiate the Data Dictionary. That function is easily added to the Contents Parser. In our sample implementation, both parsers are the same, and generation of SQL output is controlled by the setting of an internal flag variable.

Extraction of data to a usable form from the incoming data stream is a desirable property of an SDD receiver. The wide availablilty of the Java programming language, together with its machine independence, make it a prime candidate for use in the extraction problem. Using the methods described in Section 2.2.3, a provider of self-describing data can construct an extractor with Java and include the code in the Data Dictionary. Those methods require that the recipient extract the Java code and place it in operation. We are planning to extend the SDD types so that an extractor can be transmitted in Java byte code form, allowing a receiver (if implemented in Java) to automatically use the extractor to process incoming data.

3. Conclusions

In this report we present a methodology that provides a framework to create, encode, and decode a self-describing data stream. Clients of these data streams can interpret the detailed information in the data transfer with limited a priori information. We demonstrate that a self-describing data transfer be completed using only

1. existing data description language standards,
2. parsers to enforce language compliance,
3. a simple content language that flows out of the data description language, and
4. architecture neutral encoders and decoders based on ASN.1.

We demonstrate the use of the SDD paradigm with data from a legacy traffic management center, and in Chapter 2 we provide a detailed technical description of its implementation. This SDD paradigm has the potential to enhance the NTCIP family of protocols under development to support ITS deployment.

The complete source code for our implementation can be obtained by ftp from ftp://ftp.its.washington.edu/pub/mdi/SDD/v2.0.0/sdd2.0.0b6.zip, and additional information is available from http://www.its.washington.edu/bbone.

Appendix A: Schema Parser

This appendix contains the **yacc** specification of the language recognized by the Schema Parser. The action routines have been removed for clarity.

```
%token BIT
%token CHAR
%token CHARACTER
%token COMMA
%token CREATE
%token DATE
%token DAY
%token DEC
%token DECIMAL
%token DEFAULT
%token DELIMITED_IDENTIFIER
%token DOUBLE
%token FLOAT
%token FOREIGN
%token HOUR
%token INT
%token INTEGER
%token INTERVAL
%token KEY
%token LFT_PAREN
%token MINUTE
%token MONTH
%token NON_QUOTE_CHAR
%token NOT
%token NULL_TOKEN
%token NUMERIC
%token PERIOD
%token PRECISION
%token PRIMARY
%token QUOTE
%token REAL
%token REFERENCES
%token REGULAR_IDENTIFIER
%token RESERVED_WORD
%token RT_PAREN
%token SCHEMA
%token SECOND
%token SEMICOLON
%token SMALLINT
%token TABLE
%token TIME
%token TIMESTAMP
%token TO
%token UNDERSCORE
%token UNIQUE
%token UNSIGNED_INTEGER
```

```
%token VARCHAR
%token VARYING
%token WITH
%token YEAR
%token ZONE

%%
data_dictionary_definition :
      SQL_schema_statement optional_semicolon
    ;

optional_semicolon : /* empty */
    | SEMICOLON
    ;

SQL_schema_statement : schema_definition ;

schema_definition : CREATE SCHEMA schema_element_list ;

schema_element_list : /* empty */
    | schema_element_list schema_element
    ;

schema_element : table_definition ;

table_definition : CREATE TABLE table_name table_element_list ;

table_name : identifier ;

qualified_table_name : qualified_name ;

qualified_name :
      identifier PERIOD identifier PERIOD identifier
    | identifier PERIOD identifier
    | identifier
    ;

identifier :
      REGULAR_IDENTIFIER
    | DELIMITED_IDENTIFIER
    ;

table_element_list : LFT_PAREN table_element table_element_list_body
RT_PAREN ;

table_element_list_body : /* empty */
    | table_element_list_body COMMA table_element
    ;

table_element :
      column_definition
    | table_constraint_definition
    ;
```

```
column_definition :
      column_name data_type default_clause column_constraint_definition
    ;

column_name : identifier ;

data_type :
      character_string_type
    | bit_string_type
    | numeric_type
    ;

character_string_type :
      CHARACTER
    | CHARACTER LFT_PAREN length RT_PAREN
    | CHAR LFT_PAREN length RT_PAREN
    | CHAR
    ;

length : UNSIGNED_INTEGER ;

bit_string_type :
      BIT
    | BIT LFT_PAREN length RT_PAREN
    ;

numeric_type :
      exact_numeric_type
    | approximate_numeric_type
    ;

exact_numeric_type :
      NUMERIC precision_spec
    | DECIMAL precision_spec
    | DEC precision_spec
    | INTEGER
    | INT
    | SMALLINT
    ;

precision_spec : /* empty */
    | LFT_PAREN precision RT_PAREN
    | LFT_PAREN precision COMMA scale RT_PAREN
    ;

precision : UNSIGNED_INTEGER ;

scale : UNSIGNED_INTEGER ;

approximate_numeric_type :
      FLOAT
    | FLOAT LFT_PAREN precision RT_PAREN
    | REAL
    | DOUBLE PRECISION
```

```
    ;

default_clause : /* empty */
    | DEFAULT default_option
    ;

default_option : NULL_TOKEN ;

column_constraint_definition : /* empty */
    | column_constraint_definition column_constraint
    ;

column_constraint :
      NOT NULL_TOKEN
    | UNIQUE
    | PRIMARY KEY
    ;

table_constraint_definition : table_constraint ;

table_constraint :
      unique_constraint_definition
    | referential_constraint_definition
    ;

unique_constraint_definition :
      UNIQUE LFT_PAREN unique_column_list RT_PAREN
    | PRIMARY KEY LFT_PAREN unique_column_list RT_PAREN
    ;

unique_column_list : local_column_name_list ;

column_name_list : constraint_column_name column_name_list_body ;

column_name_list_body : /* empty */
    | column_name_list_body COMMA constraint_column_name
    ;

local_column_name_list : column_name local_column_name_list_body ;

local_column_name_list_body : /* empty */
    | local_column_name_list_body COMMA constraint_column_name
    ;

constraint_column_name : column_name ;

constraint_table_name : qualified_table_name ;

referential_constraint_definition :
      FOREIGN KEY LFT_PAREN referencing_columns RT_PAREN
          references_specification
    ;

referencing_columns : local_column_name_list ;
```

```
reference_column_list : column_name_list ;

references_specification : REFERENCES referenced_table_and_columns ;

referenced_table_and_columns :
      qualified_table_name
    | qualified_table_name LFT_PAREN reference_column_list RT_PAREN
    ;
%%
```

Appendix B: Schema Scanner

This appendix contains the **flex** specification of the scanner used with the Schema Parser defined in the preceeding **yacc** grammar. The scanner is called by the parser, and recognizes the tokens defined in the Schema Grammar, returning an appropriate value to the Parser.

```
A [Aa]
B [Bb]
C [Cc]
D [Dd]
E [Ee]
F [Ff]
G [Gg]
H [Hh]
I [Ii]
J [Jj]
K [Kk]
L [Ll]
M [Mm]
N [Nn]
O [Oo]
P [Pp]
Q [Qq]
R [Rr]
S [Ss]
T [Tt]
U [Uu]
V [Vv]
W [Ww]
X [Xx]
Y [Yy]
Z [Zz]
%%
"." { SchemaLength += 1; return(PERIOD); }
"_" { SchemaLength += strlen(yytext); return(UNDERSCORE); }
"(" { SchemaLength += strlen(yytext); return(LFT_PAREN); }
")" { SchemaLength += strlen(yytext); return(RT_PAREN); }
"," { SchemaLength += strlen(yytext); return(COMMA); }
"'" { SchemaLength += strlen(yytext); return(QUOTE); }
";" { SchemaLength += strlen(yytext); return(SEMICOLON); }
" " SchemaLength += 1;
\t SchemaLength += 1;
\n { SchemaLength += 1; LineNum += 1; }
{B}{I}{T} { SchemaLength += strlen(yytext); return(BIT); }
{C}{H}{A}{R} { SchemaLength += strlen(yytext); return(CHAR); }
{C}{H}{A}{R}{A}{C}{T}{E}{R} { SchemaLength += strlen(yytext);
                        return(CHARACTER); }
{C}{R}{E}{A}{T}{E} { SchemaLength += strlen(yytext); return(CREATE);}
{D}{A}{T}{E} { SchemaLength += strlen(yytext); return(DATE); }
{D}{A}{Y} { SchemaLength += strlen(yytext); return(DAY); }
```

```
{D}{E}{C} { SchemaLength += strlen(yytext); return(DEC); }
{D}{E}{C}{I}{M}{A}{L} { SchemaLength += strlen(yytext); return(DECIMAL); }
{D}{E}{F}{A}{U}{L}{T} { SchemaLength += strlen(yytext); return(DEFAULT); }
{D}{O}{U}{B}{L}{E} { SchemaLength += strlen(yytext); return(DOUBLE); }
{F}{L}{O}{A}{T} { SchemaLength += strlen(yytext); return(FLOAT); }
{F}{O}{R}{E}{I}{G}{N} { SchemaLength += strlen(yytext); return(FOREIGN); }
{H}{O}{U}{R} { SchemaLength += strlen(yytext); return(HOUR); }
{I}{N}{T} { SchemaLength += strlen(yytext); return(INT); }
{I}{N}{T}{E}{G}{E}{R} { SchemaLength += strlen(yytext); return(INTEGER); }
{I}{N}{T}{E}{R}{V}{A}{L} { SchemaLength += strlen(yytext);
return(INTERVAL); }
{K}{E}{Y} { SchemaLength += strlen(yytext); return(KEY); }
{M}{I}{N}{U}{T}{E} { SchemaLength += strlen(yytext); return(MINUTE); }
{M}{O}{N}{T}{H} { SchemaLength += strlen(yytext); return(MONTH); }
{N}{O}{T} { SchemaLength += strlen(yytext); return(NOT); }
{N}{U}{L}{L} { SchemaLength += strlen(yytext); return(NULL_TOKEN); }
{N}{U}{M}{E}{R}{I}{C} { SchemaLength += strlen(yytext); return(NUMERIC); }
{P}{R}{E}{C}{I}{S}{I}{O}{N} { SchemaLength += strlen(yytext);
                         return(PRECISION); }
{P}{R}{I}{M}{A}{R}{Y} { SchemaLength += strlen(yytext); return(PRIMARY); }
{R}{E}{A}{L} { SchemaLength += strlen(yytext); return(REAL); }
{R}{E}{F}{E}{R}{E}{N}{C}{E}{S} { SchemaLength += strlen(yytext);
                          return(REFERENCES); }
{S}{C}{H}{E}{M}{A} { SchemaLength += strlen(yytext); return(SCHEMA); }
{S}{E}{C}{O}{N}{D} { SchemaLength += strlen(yytext); return(SECOND); }
{S}{M}{A}{L}{L}{I}{N}{T} { SchemaLength += strlen(yytext);
return(SMALLINT); }
{T}{A}{B}{L}{E} { SchemaLength += strlen(yytext); return(TABLE); }
{T}{I}{M}{E} { SchemaLength += strlen(yytext); return(TIME); }
{T}{I}{M}{E}{S}{T}{A}{M}{P} { SchemaLength += strlen(yytext);
                         return(TIMESTAMP); }
{T}{O} { SchemaLength += strlen(yytext); return(TO); }
{U}{N}{I}{Q}{U}{E} { SchemaLength += strlen(yytext); return(UNIQUE); }
{V}{A}{R}{C}{H}{A}{R} { SchemaLength += strlen(yytext); return(VARCHAR); }
{V}{A}{R}{Y}{I}{N}{G} { SchemaLength += strlen(yytext); return(VARYING); }
{W}{I}{T}{H} { SchemaLength += strlen(yytext); return(WITH); }
{Y}{E}{A}{R} { SchemaLength += strlen(yytext); return(YEAR); }
{Z}{O}{N}{E} { SchemaLength += strlen(yytext); return(ZONE); }

{A}{B}{S}{O}{L}{U}{T}{E}                                  |
{A}{C}{T}{I}{O}{N}                                        |
{A}{D}{D}                                                 |
{A}{L}{L}                                                 |
{A}{L}{L}{O}{C}{A}{T}{E}                                  |
{A}{L}{T}{E}{R}                                           |
{A}{N}{D}                                                 |
{A}{N}{Y}                                                 |
{A}{R}{E}                                                 |
{A}{S}                                                    |
{A}{S}{C}                                                 |
{A}{S}{S}{E}{R}{T}{I}{O}{N}                               |
{A}{T}                                                    |
{A}{U}{T}{H}{O}{R}{I}{Z}{A}{T}{I}{O}{N}                   |
{A}{V}{G}                                                 |
```

```
{B}{E}{G}{I}{N}                                      |
{B}{E}{T}{W}{E}{E}{N}                                |
{B}{I}{T}_{L}{E}{N}{G}{T}{H}                         |
{B}{O}{T}{H}                                         |
{B}{Y}                                               |
{C}{A}{S}{C}{A}{D}{E}                                |
{C}{A}{S}{C}{A}{D}{E}{D}                             |
{C}{A}{S}{E}                                         |
{C}{A}{S}{T}                                         |
{C}{A}{T}{A}{L}{O}{G}                                |
{C}{H}{A}{R}_{L}{E}{N}{G}{T}{H}                      |
{C}{H}{A}{R}{A}{C}{T}{E}{R}_{L}{E}{N}{G}{T}{H}       |
{C}{H}{E}{C}{K}                                      |
{C}{L}{O}{S}{E}                                      |
{C}{O}{A}{L}{E}{S}{C}{E}                             |
{C}{O}{L}{L}{A}{T}{E}                                |
{C}{O}{L}{L}{A}{T}{I}{O}{N}                          |
{C}{O}{L}{U}{M}{N}                                   |
{C}{O}{M}{M}{I}{T}                                   |
{C}{O}{N}{N}{E}{C}{T}                                |
{C}{O}{N}{N}{E}{C}{T}{I}{O}{N}                       |
{C}{O}{N}{S}{T}{R}{A}{I}{N}{T}                       |
{C}{O}{N}{S}{T}{R}{A}{I}{N}{T}{S}                    |
{C}{O}{N}{T}{I}{N}{U}{E}                             |
{C}{O}{N}{V}{E}{R}{T}                                |
{C}{O}{R}{R}{E}{S}{P}{O}{N}{D}{I}{N}{G}              |
{C}{O}{U}{N}{T}                                      |
{C}{R}{O}{S}{S}                                      |
{C}{U}{R}{R}{E}{N}{T}                                |
{C}{U}{R}{R}{E}{N}{T}_{D}{A}{T}{E}                   |
{C}{U}{R}{R}{E}{N}{T}_{T}{I}{M}{E}                   |
{C}{U}{R}{R}{E}{N}{T}_{T}{I}{M}{E}{S}{T}{A}{M}{P}    |
{C}{U}{R}{R}{E}{N}{T}_{U}{S}{E}{R}                   |
{C}{U}{R}{S}{O}{R}                                   |
{D}{E}{A}{L}{L}{O}{C}{A}{T}{E}                       |
{D}{E}{C}{L}{A}{R}{E}                                |
{D}{E}{F}{E}{R}{R}{A}{B}{L}{E}                       |
{D}{E}{F}{E}{R}{R}{E}{D}                             |
{D}{E}{L}{E}{T}{E}                                   |
{D}{E}{S}{C}                                         |
{D}{E}{S}{C}{R}{I}{B}{E}                             |
{D}{E}{S}{C}{R}{I}{P}{T}{O}{R}                       |
{D}{I}{A}{G}{N}{O}{S}{T}{I}{C}{S}                    |
{D}{I}{S}{C}{O}{N}{N}{E}{C}{T}                       |
{D}{I}{S}{T}{I}{N}{C}{T}                             |
{D}{O}{M}{A}{I}{N}                                   |
{D}{R}{O}{P}                                         |
{E}{L}{S}{E}                                         |
{E}{N}{D}                                            |
{E}{N}{D}\-{E}{X}{E}{C}                              |
{E}{S}{C}{A}{P}{E}                                   |
{E}{X}{C}{E}{P}{T}                                   |
{E}{X}{E}{C}{U}{T}{E}                                |
{E}{X}{I}{S}{T}{S}                                   |
```

```
{E}{X}{T}{E}{R}{N}{A}{L}                    |
{E}{X}{T}{R}{A}{C}{T}                       |
{F}{A}{L}{S}{E}                             |
{F}{E}{T}{C}{H}                             |
{F}{I}{R}{S}{T}                             |
{F}{O}{R}                                   |
{F}{O}{U}{N}{D}                             |
{F}{R}{O}{M}                                |
{F}{U}{L}{L}                                |
{G}{E}{T}                                   |
{G}{L}{O}{B}{A}{L}                          |
{G}{O}                                      |
{G}{O}{T}{O}                                |
{G}{R}{A}{N}{T}                             |
{G}{R}{O}{U}{P}                             |
{H}{A}{V}{I}{N}{G}                          |
{I}{D}{E}{N}{T}{I}{T}{Y}                    |
{I}{M}{M}{E}{D}{I}{A}{T}{E}                 |
{I}{N}                                      |
{I}{N}{D}{I}{C}{A}{T}{O}{R}                 |
{I}{N}{I}{T}{I}{A}{L}{L}{Y}                 |
{I}{N}{N}{E}{R}                             |
{I}{N}{P}{U}{T}                             |
{I}{N}{S}{E}{N}{S}{I}{T}{I}{V}{E}           |
{I}{N}{S}{E}{R}{T}                          |
{I}{N}{T}{E}{R}{S}{E}{C}{T}                 |
{I}{N}{T}{O}                                |
{I}{S}                                      |
{I}{S}{O}{L}{A}{T}{I}{O}{N}                 |
{J}{O}{I}{N}                                |
{L}{A}{N}{G}{U}{A}{G}{E}                    |
{L}{A}{S}{T}                                |
{L}{E}{A}{D}{I}{N}{G}                       |
{L}{E}{F}{T}                                |
{L}{E}{V}{E}{L}                             |
{L}{I}{K}{E}                                |
{L}{O}{C}{A}{L}                             |
{L}{O}{W}{E}{R}                             |
{M}{A}{T}{C}{H}                             |
{M}{A}{X}                                   |
{M}{I}{N}                                   |
{M}{O}{D}{U}{L}{E}                          |
{N}{A}{M}{E}{S}                             |
{N}{A}{T}{I}{O}{N}{A}{L}                    |
{N}{A}{T}{U}{R}{A}{L}                       |
{N}{C}{H}{A}{R}                             |
{N}{E}{X}{T}                                |
{N}{O}                                      |
{N}{U}{L}{L}{I}{F}                          |
{O}{C}{T}{E}{T}_{L}{E}{N}{G}{T}{H}          |
{O}{F}                                      |
{O}{N}                                      |
{O}{N}{L}{Y}                                |
{O}{P}{E}{N}                                |
```

```
{O}{P}{T}{I}{O}{N}
{O}{R}
{O}{R}{D}{E}{R}
{O}{U}{T}{E}{R}
{O}{U}{T}{P}{U}{T}
{O}{V}{E}{R}{L}{A}{P}{S}
{P}{A}{D}
{P}{A}{R}{T}{I}{A}{L}
{P}{O}{S}{I}{T}{I}{O}{N}
{P}{R}{E}{P}{A}{R}{E}
{P}{R}{E}{S}{E}{R}{V}{E}
{P}{R}{I}{O}{R}
{P}{R}{I}{V}{I}{L}{E}{G}{E}{S}
{P}{R}{O}{C}{E}{D}{U}{R}{E}
{P}{U}{B}{L}{I}{C}
{R}{E}{A}{D}
{R}{E}{L}{A}{T}{I}{V}{E}
{R}{E}{S}{T}{R}{I}{C}{T}
{R}{E}{V}{O}{K}{E}
{R}{I}{G}{H}{T}
{R}{O}{L}{L}{B}{A}{C}{K}
{R}{O}{W}{S}
{S}{C}{R}{O}{L}{L}
{S}{E}{C}{T}{I}{O}{N}
{S}{E}{L}{E}{C}{T}
{S}{E}{S}{S}{I}{O}{N}
{S}{E}{S}{S}{I}{O}{N}_{U}{S}{E}{R}
{S}{E}{T}
{S}{I}{Z}{E}
{S}{O}{M}{E}
{S}{P}{A}{C}{E}
{S}{Q}{L}
{S}{Q}{L}{C}{O}{D}{E}
{S}{Q}{L}{E}{R}{R}{O}{R}
{S}{Q}{L}{S}{T}{A}{T}{E}
{S}{U}{B}{S}{T}{R}{I}{N}{G}
{S}{U}{M}
{S}{Y}{S}{T}{E}{M}_{U}{S}{E}{R}
{T}{E}{M}{P}{O}{R}{A}{R}{Y}
{T}{H}{E}{N}
{T}{I}{M}{E}{Z}{O}{N}{E}_{H}{O}{U}{R}
{T}{I}{M}{E}{Z}{O}{N}{E}_{M}{I}{N}{U}{T}{E}
{T}{R}{A}{I}{L}{I}{N}{G}
{T}{R}{A}{N}{S}{A}{C}{T}{I}{O}{N}
{T}{R}{I}{M}
{T}{R}{U}{E}
{U}{N}{I}{O}{N}
{U}{N}{K}{N}{O}{W}{N}
{U}{P}{D}{A}{T}{E}
{U}{P}{P}{E}{R}
{U}{S}{A}{G}{E}
{U}{S}{E}{R}
{U}{S}{I}{N}{G}
{V}{A}{L}{U}{E}
```

```
{V}{A}{L}{U}{E}{S}                                    |
{V}{I}{E}{W}                                          |
{W}{H}{E}{N}                                          |
{W}{H}{E}{N}{E}{V}{E}{R}                              |
{W}{H}{E}{R}{E}                                       |
{W}{O}{R}{K}                                          |
{W}{R}{I}{T}{E}  { SchemaLength += strlen(yytext); return(RESERVED_WORD); }

[A-Z]([A-Z0-9_]*[A-Z0-9])*  { SchemaLength += strlen(yytext);
                        return(REGULAR_IDENTIFIER); }
\"([^"\n]*(\"\")*[^"\n]*)*\"  { SchemaLength += strlen(yytext);
                        return(DELIMITED_IDENTIFIER); }
\-\-[^\n]*\n { SchemaLength += strlen(yytext); LineNum += 1; }
[0-9]+ { SchemaLength += strlen(yytext); return(UNSIGNED_INTEGER); }
[^'] { SchemaLength += strlen(yytext); return(NON_QUOTE_CHAR); }
```

Appendix C: Contents Parser

This appendix contains the **yacc** specification for the contents language.

```
%token CATALOG
%token SCHEMA
%token TABLE
%token COLUMN
%token LFT_PAREN
%token RT_PAREN
%token SEMICOLON
%token COMMA
%token NULL_TOKEN
%token DEFAULT
%token INTEGER_LITERAL
%token EXACT_DOUBLE_LITERAL
%token APPROXIMATE_DOUBLE_LITERAL
%token REGULAR_IDENTIFIER
%token DELIMITED_IDENTIFIER
%token CHARACTER_STRING
%start data_dictionary_file
%%
data_dictionary_file : table_list ;

table_list : /* empty */
    | table_list table_entry
    ;

table_entry : TABLE table_name column_list row_values_list ;

column_list : COLUMN LFT_PAREN column_name col_name_list RT_PAREN ;

table_name : identifier ;

column_name : identifier ;

identifier :
      REGULAR_IDENTIFIER
    | DELIMITED_IDENTIFIER
    ;

col_name_list : /* empty */
    | col_name_list COMMA column_name
    ;

row_values_list : /* empty */
    | row_values_list row_values
    ;

row_values :
      row_value SEMICOLON
    | row_value COMMA row_values
```

```
    ;

row_value :
      NULL_TOKEN
    | DEFAULT
    | INTEGER_LITERAL
    | EXACT_DOUBLE_LITERAL
    | APPROXIMATE_DOUBLE_LITERAL
    | CHARACTER_STRING
    ;
%%
```

Appendix D: Contents Scanner

This appendix contains the **flex** specification for the scanner used with the Contents Parser.

```
D           [0-9]
E           [eE][-+]?{D}+
Q           [']['']
%%
"(" { ContentLength += strlen(ddftext); return(LFT_PAREN); }

")" { ContentLength += strlen(ddftext); return(RT_PAREN); }

"," { ContentLength += strlen(ddftext); return(COMMA); }

";" { ContentLength += strlen(ddftext); return(SEMICOLON); }

" " ContentLength += strlen(ddftext);

\t ContentLength += strlen(ddftext);

\n { ContentLength += strlen(ddftext); LineCount += 1; }

[Nn][Uu][Ll][Ll] { ContentLength += strlen(ddftext); return(NULL_TOKEN); }

[Dd][Ee][Ff][Aa][Uu][Ll][Tt] { ContentLength += strlen(ddftext);
                        return(DEFAULT); }

[Tt][Aa][Bb][Ll][Ee] { ContentLength += strlen(ddftext); return(TABLE); }

[Cc][Oo][Ll][Uu][Mm][Nn] { ContentLength += strlen(ddftext);
return(COLUMN); }

[A-Z]([A-Z0-9_]*[A-Z0-9])* { ContentLength += strlen(ddftext);
                      return(REGULAR_IDENTIFIER); }

\"([^"\n]*(\"\")*[^"\n]*)*\" { ContentLength += strlen(ddftext);
                        return(DELIMITED_IDENTIFIER); }

\-\-[^\n]*\n { ContentLength += strlen(ddftext); LineCount += 1; }

[+\-]?{D}+ { ContentLength += strlen(ddftext); return(INTEGER_LITERAL); }

[+\-]?{D}+[.] { ContentLength += strlen(ddftext);
return(EXACT_DOUBLE_LITERAL); }

[+\-]?{D}+[.]{D}+ { ContentLength += strlen(ddftext);
              return(EXACT_DOUBLE_LITERAL); }

[+\-]?[.]{D}+ { ContentLength += strlen(ddftext);
return(EXACT_DOUBLE_LITERAL); }

[+\-]?{D}+{E} { ContentLength += strlen(ddftext);
```

```
                    return(APPROXIMATE_DOUBLE_LITERAL); }

[+\-]?{D}+[.]{E} { ContentLength += strlen(ddftext);
                    return(APPROXIMATE_DOUBLE_LITERAL); }

[+\-]?{D}+[.]{D}+{E} { ContentLength += strlen(ddftext);
                        return(APPROXIMATE_DOUBLE_LITERAL); }

[+\-]?[.]{D}+{E} { ContentLength += strlen(ddftext);
                    return(APPROXIMATE_DOUBLE_LITERAL); }

([`])([^'\n]*|[`][`])*([`]) { ContentLength += strlen(ddftext);
               return(CHARACTER_STRING); }
```

Appendix E: Schema Definition

This appendix contains the definition of the Data Dictionary schema used by the SDD transmitter described in Chapter 2.

```
CREATE SCHEMA — recipient will need to supply AUTHORIZATION information

CREATE TABLE CABINETS
(CABINET_ID CHAR(7)    NOT NULL PRIMARY KEY,
 FREEWAY    CHAR(10)   NOT NULL,
 TEXT       CHAR(255),
 RAMP       SMALLINT   NOT NULL)

CREATE TABLE LOOPS
(LOOP_ID     CHAR(16)   NOT NULL PRIMARY KEY,
 CABINET_ID  CHAR(7)    NOT NULL,
 METERED     CHAR(4)    NOT NULL,
 RD_TYPE     CHAR(30)   NOT NULL,
 DIRECTION   CHAR(12)   NOT NULL,
 LANE_TYPE   CHAR(20)   NOT NULL,
 LANE_NUM    SMALLINT,
 SENSOR_TYPE CHAR(12)   NOT NULL,
 DATA_OFFSET INTEGER    NOT NULL)

CREATE TABLE COORDINATES
(COORD_TYPE CHAR(40) NOT NULL PRIMARY KEY,
 NAME1      CHAR(40) NOT NULL,
 NAME2      CHAR(40) NOT NULL,
 NAME3          CHAR(40) NOT NULL,
 UNIT1          CHAR(40) NOT NULL,
 UNIT2          CHAR(40) NOT NULL,
 UNIT3          CHAR(40) NOT NULL)

CREATE TABLE CABINET_LOCATION
(CABINET_ID CHAR(7)    NOT NULL,
 COORD_TYPE CHAR(40)   NOT NULL,
 AUTHORITY  CHAR(30)   NOT NULL,
 VALUE1         DEC(11,8),
 VALUE2         DEC(11,8),
 VALUE3         DEC(11,8),
 PRIMARY KEY (CABINET_ID, COORD_TYPE, AUTHORITY),
 FOREIGN KEY (COORD_TYPE, AUTHORITY) REFERENCES MEASURES)

CREATE TABLE MEASURES
(COORD_TYPE CHAR(40) NOT NULL,
 AUTHORITY  CHAR(30) NOT NULL,
 REF_SYSTEM CHAR(128) NOT NULL,
 REF_PT1    DEC(11,8),
 REF_PT2    DEC(11,8),
 REF_PT3    DEC(11,8),
 ACCURACY1  DEC(11,8),
```

```
 ACCURACY2  DEC(11,8),
 ACCURACY3  DEC(11,8),
 PRIMARY KEY (COORD_TYPE, AUTHORITY),
 FOREIGN KEY (COORD_TYPE) REFERENCES COORDINATES)

CREATE TABLE LOOP_DATA
(SENSOR_ID        CHAR(15) NOT NULL,
 DATA_TIME        CHAR(30)  NOT NULL,
 VOLUME           SMALLINT  NOT NULL,
 SCAN_COUNT       SMALLINT  NOT NULL,
 FLAG             SMALLINT  NOT NULL,
 LANE_COUNT       SMALLINT  NOT NULL,
 INCIDENT_DETECT SMALLINT  NOT NULL,
 PRIMARY KEY (SENSOR_ID, DATA_TIME))

CREATE TABLE STATION_DATA
(SENSOR_ID        CHAR(15) NOT NULL,
 DATA_TIME        CHAR(30)  NOT NULL,
 VOLUME           SMALLINT  NOT NULL,
 SCAN_COUNT       SMALLINT  NOT NULL,
 FLAG             SMALLINT  NOT NULL,
 LANE_COUNT       SMALLINT  NOT NULL,
 INCIDENT_DETECT SMALLINT  NOT NULL,
 PRIMARY KEY (SENSOR_ID, DATA_TIME))

CREATE TABLE SPEED_TRAP_DATA
(SENSOR_ID        CHAR(15)  NOT NULL,
 DATA_TIME        CHAR(30)  NOT NULL,
 SPEED            DEC(4,1)  NOT NULL,
 LENGTH           DEC(4,1)  NOT NULL,
 FLAGS1           SMALLINT  NOT NULL,
 FLAGS2           SMALLINT  NOT NULL,
 BIN1             SMALLINT  NOT NULL,
 BIN2             SMALLINT  NOT NULL,
 BIN3             SMALLINT  NOT NULL,
 BIN4             SMALLINT  NOT NULL,
 PRIMARY KEY (SENSOR_ID, DATA_TIME))

CREATE TABLE LOOP_FLAGS
(FLAG_VAL         SMALLINT  NOT NULL,
 EXPLANATION      CHAR(24)  NOT NULL,
 PRIMARY KEY (FLAG_VAL))

CREATE TABLE STATION_FLAGS
(FLAG_VAL         SMALLINT  NOT NULL,
 EXPLANATION      CHAR(24)  NOT NULL,
 PRIMARY KEY (FLAG_VAL))

CREATE TABLE INCIDENT_DETECT
(FLAG_VAL         SMALLINT  NOT NULL,
 EXPLANATION      CHAR(24)  NOT NULL,
 PRIMARY KEY (FLAG_VAL))
```

```
— To deal with lack of a "text" data type in SQL, we use the construct
— that follows:
— The lines are stored in a table, keyed by line number and
— measure name.
— To get the text for a given file, select all lines with the
— desired FILE_NAME, in LINE_NUMBER order.

CREATE TABLE ALG_DESCRIPT — Algorithms used to extract data from
                          — the real data stream.
(FILE_NAME CHAR(16) NOT NULL,
 LINE_NUMBER  INTEGER  NOT NULL,
 LINE_TEXT    CHAR(125),
 PRIMARY KEY (FILE_NAME, LINE_NUMBER))
;
```

Appendix F: Contents File

This appendix contains an abbreviated version of the Dictionary Contents file created when the SDD transmitter described in Chapter 2 receives a legacy dictionary. Since the complete file is approximately 450 kilobytes in size, this version is heavily edited.

```
TABLE ALG_DESCRIPT
COLUMN (FILE_NAME, LINE_NUMBER, LINE_TEXT)
'TmsLoop.java', 0, '' ;
'TmsLoop.java', 1, '' ;
'TmsLoop.java', 2, '/**' ;
'TmsLoop.java', 3,
' * Encapsulates a TMS loop sensor in packed format, plus other fields' ;
'TmsLoop.java', 4,
' * necessary to support SDD.  This is an immutable class which may' ;
'TmsLoop.java', 5,
' * only be initialized from a TMS data block.' ;

... the remainder of the TmsLoop.java file

'TmsStation.java', 0, '' ;
'TmsStation.java', 1, '' ;
'TmsStation.java', 2, '/**' ;
'TmsStation.java', 3, ' * Encapsulates a TMS station in packed format' ;
'TmsStation.java', 4, ' */' ;
'TmsStation.java', 5, '' ;
'TmsStation.java', 6, '' ;
'TmsStation.java', 7, 'public class TmsStation extends TmsData' ;
'TmsStation.java', 8, '{' ;
'TmsStation.java', 9, '  /**' ;
'TmsStation.java', 10,
'   * Constructs a Tmsstation using the 3 bytes of data at' ;

... the remainder of the TmsStation.java file

'TmsTrap.java', 0, '' ;
'TmsTrap.java', 1, '/**' ;
'TmsTrap.java', 2,
' * Encapsulates a TMS speed trap in packed format.  Data covers ' ;
'TmsTrap.java', 3, ' * a period of 20 seconds.' ;
'TmsTrap.java', 4, ' *' ;

... the remainder of the TmsTrap.java file

TABLE COORDINATES
COLUMN (COORD_TYPE,NAME1,NAME2,NAME3,UNIT1,UNIT2,UNIT3)
'spherical', 'x', 'y', 'z',
             'kilometers', 'kilometers', 'kilometers' ;
'geodetic', 'latitude', 'longitude', 'altitude',
            'degrees', 'degrees', 'feet' ;
```

```
'state plane', 'x', 'y', 'not used',
                'miles', 'miles', 'not used' ;
'linear', 'mile marker', 'not used', 'not used',
          'miles', 'not used', 'not used' ;

TABLE MEASURES
COLUMN (COORD_TYPE, AUTHORITY, REF_SYSTEM,
        REF_PT1, REF_PT2, REF_PT3,
        ACCURACY1, ACCURACY2, ACCURACY3)
'geodetic', 'WSDOT', 'NAD23', 0, 0, 0, 0.001, 0.001, 0.001 ;
'geodetic', 'UW ITS group', 'NAD89', 0, 0, 0, 0.001, 0.001, 0.001 ;
'linear','WSDOT GIS Section','WSDOT 1997', 0, NULL, NULL, 0.01, NULL, NULL
;
'linear','TMC RTDB','WSDOT 1997', 0, NULL, NULL, 0.01, NULL, NULL ;

TABLE LOOP_FLAGS
COLUMN ( FLAG_VAL, EXPLANATION )
0, 'Good Data' ;
1, 'Short Pulse' ;
2, 'Chatter' ;
3, 'Outside Vol/Occ Envelope' ;
4, 'Reserved' ;
5, 'Reserved' ;
6, 'Operator Disabled' ;
7, 'Bad Loop' ;

TABLE STATION_FLAGS
COLUMN ( FLAG_VAL, EXPLANATION )
0, 'Data not Usable' ;
1, 'Data Usable' ;

TABLE INCIDENT_DETECT
COLUMN ( FLAG_VAL, EXPLANATION )
0, 'No Incident' ;
1, 'Tentative' ;
2, 'Occurred' ;
3, 'Continuing' ;

TABLE CABINETS
COLUMN ( CABINET_ID, FREEWAY, TEXT, RAMP )
'ES-059D', 'I-5', '"S170thSt"', 0 ;
'ES-068D', 'I-5', '"S154thSt"', 0 ;
'ES-069D', 'UNKNOWN', 'UNKNOWN', 0 ;
'ES-074D', 'I-5', '"S129thSt"', 0 ;
'ES-079D', 'I-5', '"SNorfolkSt"', 0 ;

... one entry for each cabinet in the legacy data dictionary.

TABLE CABINET_LOCATION
COLUMN ( CABINET_ID, COORD_TYPE, AUTHORITY, VALUE1, VALUE2, VALUE3 )
'ES-059D','linear','TMC RTDB',153.51, NULL, NULL ;
'ES-059D','linear','WSDOT GIS Section',153.510000, NULL, NULL ;
'ES-059D','geodetic','WSDOT',47.449, -122.267, NULL ;
```

```
... one entry for each set of coordinates (possibly several per cabinet).

TABLE LOOPS
COLUMN ( LOOP_ID, CABINET_ID, METERED, RD_TYPE, DIRECTION,
         LANE_TYPE, LANE_NUM, SENSOR_TYPE, DATA_OFFSET )
'ES-059D:_MNH__5', 'ES-059D', 'NO', 'mainline', 'northbound',
'HOV mainline', 5, 'loop', 16 ;
'ES-059D:_MN_Stn', 'ES-059D', 'NO', 'mainline', 'northbound',
'mainline', 0, 'station', 19 ;

... one entry for each loop in the legacy dictionary.
```

Appendix G: Newsletter of the ITS Cooperative Deployment Network

Describing "Self-Describing Data"

Looking Under the Hood at the Seattle MDI Data Engine

(http://www.nawgits.com/sdd_dd.html)

The Seattle Smart Trek *(http://www.smarttrek.org/)* Project, perhaps more than any other Model Deployment Initiative *(http://www.its.dot.gov/mdi/)* (MDI) project, includes the participation of a wide range of private-sector firms as so-called "Information Service Providers" or "ISPs." These ISPs often take raw data about traffic conditions (typically from loop or video detectors) and then "add value" by post-processing that data and communicating it to travelers through a variety of channels and devices. A key question in that scenario: how best to acquire that data while requiring a minimum of hand-holding on the part of its source (in this case, the Washington State Department of Transportation)? Many of the Smart Trek projects have found the answer in so-called "self-describing data," in which the data stream itself contains not only raw traffic data but information about where the data comes from and what it means. ICDN Editor Jerry Werner recently discussed the background and potential national applicability of SDD with its developer and foremost proponent, Professor Dan Dailey of the University of Washington. Dailey, the director of UW's Intelligent Transportation Systems *(http://www.its.washington.edu/)* program, is a research track Associate Professor in the Department of Electrical Engineering who also holds adjunct appointments in the departments of Civil Engineering and Technical Communication in the College of Engineering.

The Discussion

ICDN: How long have you been involved with ITS projects?

Dailey: My relationship to using traffic data in modeling goes back to 1989. My involvement was initially modeling using data from the Washington State Department of Transportation traffic management center. The present ITS backbone activity evolved as sort of a defense mechanism. We had a series of students who wanted to do modeling. Each student that came along said 'well, how do I get the data?' and invented a new way to do so. Eventually, we came up with the idea for reusable code to make the data available and not spend the first six months of a project redoing the data acquisition task. Instead, students can spend time actually doing something with the data. So, we created a set of Unix applications that were designed to extract data from remote places, in this case the Washington State Dept. of Transportation's traffic management center. We obtained data from loop sensors for about 3,000 locations around the region's highways. We can measure something about the duration of time a vehicle is over the loop and from that do modeling. At about the same time, we also learned that the people at Metro Transit, the local transit operator, put a vehicle location system in operation — a way to track their buses. A student took a "sniffer" and eavesdropped on their network in their building downtown. He found that we could collect packets off their network containing data about how often the bus wheels turn and bring that data back to the UW to do modeling.

ICDN: Sounds like your students will be well prepared for a career in espionage!

Dailey: You use what tools you have available. Actually, espionage is perhaps an apt term, because what you have to do to obtain data from many agencies is essentially espionage. The agencies have a funded mission directed toward doing a particular task, and that mission really has nothing to do with providing data to "those guys at the University" for research. They're not going to allocate staff resources or any dollars or time to support the University if it detracts from their mission. They'll let you put your computer under the coffeepot, and if you can hook up to the wire and it doesn't cost them anything but the electricity to run your computer — that's okay!

ICDN: Did Washington State DOT fund this early work?

Dailey: Yes, WSDOT and the USDOT gave me little bits of funding, starting at about $21,000-a-year, several years ago, which was steadily built up by a series of additional projects. The ITS notion of combining data from a lot of places and making products downstream of it was becoming popular then. We had already been in the business of pulling the data out of agencies to do these smaller projects. We were already doing what needed to be done to build a backbone.Actually, I had attended a number of meetings as early as 1992 and suggested that most regions need a backbone to make modeling and applications possible in more communities. That idea has had a mixed appeal: It has a positive appeal to public-sector transportation agencies in terms of potentially creating a level playing field, but it has a negative appeal for commercial entities — that would like to control that data.

ICDN: Do companies have an interest in pulling out data through some sort of backbone, but not of putting new information in?

Dailey: Yes, they have an interest in controlling data access, as information is potentially money. There is this notion that if you're the only company that can sell data to others — you have a contract to be the single source — that you can then charge whatever fees you want. Conversely, if your vision is for a State to have an obligation to create a level playing field among commercial information providers, you would want to come up with a set of paradigms and protocols to effectively give away that data in a clearly understandable format. That is the "self-described data" concept, which largely eliminates the need to then answer downstream questions about the data. We used to receive requests for traffic data from Research Triangle (North Carolina) and California among others, and we would provide that data to them. Then they'd call back the next week and want one of our programmers to consult with them for a month to explain the data to them. We didn't have that resource. So we created a protocol called self-describing data that in theory addressed that problem.

ICDN: Self-describing data is being used on a number Smart Trek projects, right?

Dailey: Yes. The idea is that you need to get one self-describing data receiver and then you can understand data from a variety of sources.

ICDN: Users tap into that data via the Internet, is that right?

Dailey: We use a set of Internet protocols because the Internet exists and it's the cheapest way to do it. You could have dedicated connections, but the Internet is the low-cost way of getting the data.

ICDN: I understand that this receiver you refer to is written in the Java language.

Dailey: Yes, we give away a Java-based receiver.

ICDN: You chose Java so it will run on different platforms?

Dailey: That is correct. We originally programmed it in C on a Unix platform, but the MDI project said to us, "You have to run a Windows NT environment,' and we in turn asked, 'What version?' Their reply was, 'There are versions?' So we adopted Java in order to be platform independent.

ICDN: Which specific Smart Trek applications are using SDD right now?

Dailey: Traffic TV *(http://www.its.washington.edu/trafchan/)*, Buslink *(http://www.its.washington.edu/buslink/)*/Transit Watch *(http://www.its.washington.edu/transitwatch/)*, and Busview *(http://www.its.washington.edu/busview/)* all use it. We're giving it away to the North Seattle ATMS, which is a flow of data from WSDOT and then back into WSDOT. North Seattle ATMS is a new management system they're bringing on-line, but the easiest way for them to get the data out of their historical systems was just tapping into the backbone. It's also being used by Fastline, which markets traffic applications on hand-held, palm-top computers, and it has been used by a number of research institutions around the country.

ICDN: Is it being used yet in any other similar ITS projects, like TravInfo in San Francisco's Bay Area.

Dailey: No, not at this point. I don't know the details of the other projects, but until you have some thing that you hand them — some code you can use to build transmitters and receivers — people won't adopt it. People didn't rush up and adopt Internet protocols; they adopted the ability to do work with these protocols. So, now we're giving away the receivers and writing generic transmitters to give away in the next release from our web site. And people can use it. We've had inquiries from the other sites, but without making it so easy that they wouldn't do anything else, they're not going to adopt it beyond their specialty protocols. We were involved with Etak, Seiko, and other companies in the previous SWIFT project. At that time, it was simply much easier for them to describe a protocol that would work that year for that data and be done with it, than to think about what might happen over the next 5 years.

ICDN: Is Etak using it now?

Dailey: I don't know. They've expressed an interest in it, but I don't know if they're actually using it right now. We built a specialty protocol for them for the SWIFT project, which I believe they are still using.

ICDN: I understand that your self-describing data approach is an alternative to having a central data server, such as the one used in Atlanta for the Traveler Information Showcase. Is that a fair statement?

Dailey: Yes. There's no requirement to geographically centralize the data. Instead, the SDD protocols allow you to get data from a number of sources. You can imagine those sources would live in a number of different places geographically. What's common is the way that you get it. In a simple example, SDD is used in the same sense that you use the HTML language for web pages: you share a language between your browser and the people who are sending information. You can go to a lot of different places on the web and get that data into your browser. SDD involves a much more complex language to describe data, but it's essentially that same concept.

ICDN: Is the transmitter the piece of software that gets the data onto the network via a TCP/IP protocol?

Dailey: The transmitter is a piece of software that takes real-time legacy data, which all of the agencies produce — usually in some binary format for their own management purposes, and packages it into the SDD format. The SDD format is a combination of intelligent "meta-data" that says what the data is; the content that says what type of sensors we have, where they are located, and what they measure; and finally binary data. We talk about those three pieces on technical papers that are on-line *(http://www.its.washington.edu/its_pubs.html)*. The combination of not only the binary data but also an understanding of what it is, is what SDD gives you in completeness.

ICDN: Does the description of the data, what you call the "meta data," take up much overhead in the overall scheme of things?

Dailey: Anything that provides additional information is information rather than overhead. You can view it as information or as overhead in the same sense that if you use TCP/IP (Internet) protocol and type one character on your keyboard, you sent 43 bytes to get that character from there and back. But those 43 bytes guarantee your character will get to there and back correctly. The important thing here is that that each time the data structure changes — say, for instance, that the folks at WSDOT add new sensors on the highway — somehow you need to communicate that dynamically to everyone downstream of you so that they can use the new data. In that case, you would need to send a new data dictionary because without a data dictionary, the new data would be meaningless.

ICDN: As I understand it, you just send the data dictionary once, you don't send it periodically every so many seconds?

Dailey: You send it once when you connect because you don't know what the data is until you get the data dictionary. Then whenever the source of a data changes it, a new data dictionary is transmitted so you can understand the following data. In our case, we were working with the WSDOT people who could sit at their console and change the data once a day, once a minute, or once a month without telling us. We had no control over that process. Realistically, operators don't change things a lot. But they couldn't tell us they would do it once a day, once a month, or whatever, so the "overhead" is that you need to understand what the data is and that "overhead" takes place anytime the meaning of the data changes.

ICDN: So, what would be involved if another DOT — say Illinois DOT, because they have a policy of making data freely available to anybody who can use it — is interested in using self-describing data? How big a process is it for them to understand the SDD concept and to put their data out on the Internet and to work with ISP's that want to take it? Have you done most of the legwork already? Is SDD "plug-and-play" at this stage?

Dailey: Transmitters are the keys to that, because transmitters have to turn legacy data into generic data. We have a transmitter under testing right now; we have 2 more kinds of data coming to us, and we're going to build specific transmitters with our generic transmitter and tell folks how hard it is to do. Instructions are available right now to do that, so someone who was modestly clever could take what we have right now, hook it up to their data, and make it work.

ICDN: The transmitter needs to be customized for each site?

Dailey: Of course. Think about it. Each site has custom data; we're trying to make the transmitters generic.

ICDN: How about on the receiving end?

Dailey: The receiver we're giving away now would work for any data anywhere. So once Illinois customized our transmitters to work with their data, they could use the same receiver we're using in Seattle to receive data in Illinois.

ICDN: In reading some of your documentation, it's clear that your receiver outputs data in a raw tabular and ASCII format, but obviously a lot of post-processing would be required to do anything with that data. Is that a fair statement?

Dailey: Of course. The intent of the receiver was never actually to be "the application" The intent of the receiver was to be a framework in which you could build an application to use the data. Once users discovered that they could get these tables of data and understand what these tables meant, they didn't bother using the model to build new applications, they just used the data in a couple of cases. That's one model. Another model that Fastline uses involves taking both the data and our hooks in the databases and actually pulling the data out and putting it in their own database. Downstream, they can then do whatever they want with their own database. So, there are two models that people follow — the "quick-and-dirty model" and the "long-term investment model."

ICDN: You make the software, instructions, and documentation for self-describing data freely available, is that right?

Dailey: Yes.

ICDN: Is licensing involved?

Dailey: Yes, the whole package is Copyrighted by the University of Washington, and any transmitters we give away right now will say "For Noncommercial Purposes Only." If a company or consulting firm wants to use it in a product they sell to someone else, then we'd have to address that licensing issue. The commercial use details haven't been worked out; there's no reason to worry about that until there's money on the table.

ICDN: So no money from SDD licenses is flowing into the university at this stage?

Dailey: Our objective is to give it away in hopes that if it is adopted by public agencies it will provide a way of sharing data across the country.

ICDN: Will SDD receivers require a license or will they be royalty free?

Dailey: Well, I"d get chewed out by the intellectual property people of the university if I said they were royalty free. They are copyrighted by the University of Washington and available for use for noncommercial purposes.

ICDN: But aren"t you trying to encourage use by commercial ISPs (Note: the term ISP in this context means "information service provider") that are sprouting up around the country?

Dailey: Right. And we would happily cooperate with them. For right now, we are giving it away. We know who's using it, but we haven't stopped anyone from using it.

ICDN: Okay, but there could be some royalties involved down the road?

Dailey: There could be, but it hasn"t come up.

ICDN: Is the SDD compatible with the "National ITS System Architecture?" I know they've looked at it many times.

Dailey: Yes.

ICDN: Did it help define portions of the architecture, do you think?

Dailey: Absolutely not.

ICDN: What is its relationship with National Architecture?

Dailey: The National ITS System Architecture envisions named flows between certain types of agencies, includingTransit Management and Transportation Management Agencies. The SDD is an implementation of those flows. We're considering overlaying the naming convention that they use for the National Architecture onto the SDD scheme, so it becomes clearer where the SDD names map into the National Architecture names.

ICDN: That's probably a good idea.

Dailey: Yes, but it takes time.

ICDN: I know it does, but it might reduce some potential concern and it might make it clearer how the SDD concept fits into the national architecture.

Dailey: We're going down that path. We can't have that task take priority over what we're doing, however. We're doing the Seattle MDI project first and my national vision second because of the way the funding works out. There's funding for the MDI project and not for my national vision. But, no, the person who's working building the transmitter right now has attended the architecture classes; we have the national architecture CDs, and are working on architecture compliance. It's clearly mapped into the architecture. It will be clearer when we change our naming structure to match the architecture names. We had long discussions about that in January; however, we had a change of personnel because right now it's a hot job market out there. We are now starting those discussions over again.

ICDN: How does the SDD approach relate to NTCIP? I know you're doing a data dictionary and that NTCIP also has an ongoing data dictionary effort.

Dailey: NTCIP is a set of protocols. The ones that are implemented right now address communication between traffic control masters on the street and the agency, and changeable message signs — all what might be called "intra-agency applications." This fall, we'll be publishing a TRB paper that discusses exactly how NTCIP and SDD fit together.

ICDN: Is that paper already written?

Dailey: Yes, it's being reviewed right now. In essence, what it comes down to is that NTCIP has a number of types of defined kinds of data, and they have room for defining more kinds of data. So NTCIP can have simple data types and complex types as described by a language called "Abstract Syntax Notation.1." It's an ISO standard. Some implementations allow you to transmit that data. We actually transmit using the ASN.1 encoding of the data. NTCIP and ISO have a tree of numbers and names. So, if they'd assign us a number at the bottom of their tree saying this is one of the data types, we would then map into their data notions. The difference between the larger NTCIP efforts, which don't involve just data communication but involve dictionaries and so forth, is that there are 2 different notions that you can hold pertaining to dictionaries: One is we all go out and we buy the same dictionary. I can say 42 to you because you can look in your dictionary and know what 42 means. The other notion is that you may know in advance that your dictionary may change dynamically, that is the model we are using, and it requires transmitting a new dictionary when the data meaning changes.

ICDN: You have the ability to change, enhance, or expand the dictionary yourself at anytime, don't you?

Dailey: Right. Or contract it — whatever is necessary to make it work. We're sort of a super-set of the dictionary that everyone is holding, because if everyone hooked up and got SDD at this moment, we'd all hold the same dictionary. The models would match exactly. Until that dictionary changed, we'd all be doing NTCIP. When the dictionary changed then, the NTCIP model doesn't have any mechanism except for going through a committee process of changing the dictionary, which makes it a little harder.

ICDN: So, do you anticipate more and more discussions about the data dictionary with the NTCIP folks?

Dailey: I attended a number of meetings with NTCIP and presented our SDD ideas early on in the development of what they're doing. There's no conflict and they're well aware of our work. Nothing they're going to do will conflict with this work.

ICDN: Do they see the SDD concept as "competition" as far as you know?

Dailey: I never asked them that question. I asked them 'Are we compatible?' and 'Can we work within your domain?' We shouldn't be competing — we should be collaborating. I should stress that — this should be something that fits underneath their rubric if we're all doing our jobs.

ICDN: Is it fair to characterize you as the "granddaddy" of SDD?

Dailey: "Proud, proud papa of young baby" is a better characterization.

ICDN: Do you believe the whole concept has been proven or are there still some things to prove?

Dailey: It's a set of protocols. Proof is in the usage, in a sense. An analogy is how TCP/IP was chosen. Some years ago there existed a bunch of protocols, TCP/IP, DECNET, AppleTalk, IPX, and to this day, many of those still exist. Most folks picked TCP/IP because people from MIT gave away some implementations and eventually Microsoft sold it and then everyone used it. So, there was nothing wrong with the other protocols. They just weren't adopted as thoroughly. They are still in use but only in little corners.

ICDN: And so you'd like to see yours in that same TCP/IP model?

Dailey: We're giving it away with that same notion, that if you give something away as a first implementation of it, people will try it out and if it does what they need, they'll use it. And, of course, if it doesn't, they won't.

ICDN: What are your future plans for SDD? Where do you see it going? You mentioned your primary involvement so far has been with the WSDOT projects, right?

Dailey: Right. I've had discussions with people from other universities who are doing data communication in other science areas, such as forestry. The notion applies to any sort of dynamic data; it's not specific to transportation data. We just happened to come across it in that domain.

ICDN: Did you invent the notion itself that you combine the description with the data?

Dailey: In terms of data flow across the network, yes. In terms of earlier work, there was earlier work for self-describing databases by Leo Mark, who is now a faculty member in Maryland, I believe. I don't think he called it "self-describing data" because he had no notion of transferring data. He had the notion of creating databases that had this property. We think we invented it but you know, there's nothing new under the sun. I know that we've sent it out for publication and no one has said 'Oh, yeah, Joe did that before.'

ICDN: Have you patented it, to the extent you can patent software?

Dailey: No. It is copyrighted through the UW.

ICDN: The SDD concept theoretically means that ITS information service providers, "ISPs," could essentially provide new services in new cities very easily if they've already hooked into the data in another city, is that correct?

Dailey: That was our intent.

ICDN: So that's the allure, the attractiveness of the overall concept?

Dailey: That's our intent, but only if we can get other cities to adopt it, so that's why we're giving it away.

ICDN: As far as you know, do you think that any other cities are on the brink of adopting it, or is it still too early to tell?

Dailey: I don't have an answer to that question. We've had some discussions with New York. We've had discussions with Arizona. What it comes down to is we haven't given away a robust transmitter yet. Until we give the transmitter away and they can see that it's easy to give their data away that way rather than some other way, there's no way they can evaluate it. The transmitter is done, but it's still being tested. We'll be giving away a transmitter this fall. Call me in December — I'll be in a better position to answer that question then.

ICDN: Is the transmitter also written in Java?

Dailey: Yes, it's also in Java. We had problems doing it in other languages because certain system calls are necessary if you use C or other languages that aren't portable across platforms.

ICDN: Is there anything else germane to SDDs you can think of that would be important to people deploying systems?

Dailey: The SDD concept was originally implanted in C, later C++, then Java. So, we're not tied to a language implementation. Java was the popular language that gave us platform independence just now, but there's no reason you would have to adopt Java for your agency to be able to use this notion. I would like to separate those two things. Because we did a Java implementation, we put an object-oriented overlay on the concept of self-describing data, so that we'll be in line with the National Architecture as it moves to a more object-oriented view of the world. There's nothing inherent that has to be object-oriented about this idea — this is a concept for data transfer.

ICDN: We've talked about the fact that SDDs potentially let ISPs grab information that the public sector is putting on the Net and use it however they want. Can a number of sources combine data into one data stream?

Dailey: Yes. That is a layer above SDD that involves operating on the data and putting it back together. If you look at our SDD distribution site, you'll see a model for how data comes in and goes out. Our ITS Journal article *(http://www.its.washington.edu/pubs/its_jour.pdf)* [Note:

requires an Adobe Acrobat Reader *(http://www.adobe.com/products/acrobat/readstep.html)*] talks about an architecture for building ITS applications with reusable components. It shows a model where these components would plug together and might include something that we'd call a "data fusion element." A data fusion element might put 2 data streams together and output the sum of the streams. SDD would handle the data flow through the components that operate on that data.

ICDN: Before we close, I'd like to get back to your comments about how the private sector might view the SDD concept. You were saying that the private sector might not fully embrace the concept?

Dailey: The SDD concept is very attractive to the public sector because it provides a way of giving data away to everyone on an equal footing. However, it's a mixed bag from the private sector's standpoint. On the positive side, they could conceptually go into any new market and gain access to data quickly. On the negative side, not having control of the data in any market could be a disadvantage.

ICDN: That is a very interesting quandary.

Dailey: My understanding is that the business model for several private sector companies is to control the data in each of the market's they go into. They're going to standardize by getting a license to be the sole distributor for data. That's a very different market model than one that would create a level playing field for lots of players. However, people in the private sector have told me that they are convinced that the market won't bear open competition with a large number of players.

DESCRIPTION: The National Associations Working Group for ITS' (NAWG) ITS Cooperative Deployment Network (ICDN) is a cooperative effort to share and exchange information on Intelligent Transportation Systems (ITS) through a shared Internet resource for the benefit of NAWG members, produced by the Institute of Transportation Engineers (ITE) and its agent, Communication Alchemy, Inc, and funded by the U.S. Department of Transportation.

DISCLAIMER OF OPINIONS AND VIEWS: The views and opinions of document authors do not necessarily state or reflect those of the U.S. Government or any agency thereof or the Institute of Transportation Engineers, or Communication Alchemy, Inc., or members of the National Associations Working Group for ITS, or any members of the National Associations Working Group for ITS' ITS Cooperative Deployment Network, or any of their directors, officers, employees, agents, and members. The information provided and the views and opinions of opinions expressed in material submitted to the ICDN are those of the participant organization attributed to the material posted on the ICDN's shared Internet resource.

DISCLAIMER OF LIABILITY: Neither the U.S. Government nor any agency thereof, nor the Institute of Transportation Engineers, nor Communication Alchemy, Inc., nor members of the National Associations Working Group for ITS, nor any members of the National Associations Working Group for ITS' ITS Cooperative Deployment Network, nor any of their directors, officers, employees, agents, and members makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights.

DISCLAIMER OF ENDORSEMENT: Reference to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the U.S. Government nor any agency thereof, or the Institute of Transportation Engineers, or CommunicationAlchemy, Inc., or members of the National Associations Working Group for ITS, or any members of the National Associations Working Group for ITS' ITS Cooperative Deployment Network, or any of their directors, officers, employees, agents, and members.

This web page created by the National Associations Working Group for ITS (NAWG), a cooperative effort of organizations whose members are spearheading ITS deployment in the U.S. The NAWG makes every effort to ensure the accuracy of these pages, although errors can an do occur. Report any errors or omissions to the NAWG webmaster. Each participating member of the National Associations

Working Group for ITS is responsible only for the information it provides.

Appendix H: Self-Describing Data for Dummies

Self-Describing Data (SDD) for Dummies

Smart Trek

For Those who Would like to Understand More about SDD, but were Afraid to Ask

Version 1.4 November 17, 1997

Mark A Whiting
Smart Trek MMDI
1.509.375.2237
Mark.A.Whiting@pnl.gov

The Smart Trek SDD Development team is led by Dr. Daniel Dailey
at the Univeristy of Washington
See: http://www.its.washington.edu/bbone/ for
more information.

Self-Describing Data for Dummies

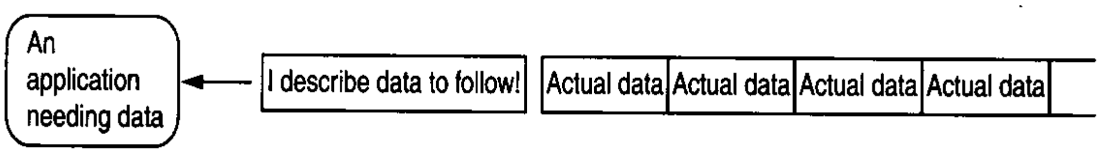
Introduction

Self-Describing Data (SDD) is an approach to transmitting and delivering data, such as transportation data, where a stream of actual data is prefixed with a set of data that "describes" it. This descriptive data can be anything. Yet the intent is to provide descriptive data that helps users, and applications, to understand the actual data that follows. An example data stream is the traffic loop sensor data collected by the Washington State Department of Transportation (WSDOT), where over two thousand sensors provide new data every twenty seconds. The descriptive data for this stream consists of descriptions of the sensors, their cabinets, their locations, and so on. This preface plus the continuous actual data stream create an SDD stream.

SDD is often difficult to understand as it is a new approach to data packaging and transmission and because the approach is still evolving. This document attempts to provide an introduction to SDD for those of us trying to keep up.

What is Self-Describing Data?

Self-Describing Data (SDD) is a data stream that consists of descriptive data, followed by a continuous stream of data. Figuratively, an SDD stream would appear as:



In terms of traffic data, this stream might appear (using English prose):

> "You'll be getting traffic loop sensor data; that belongs to these cabinets X, Y, Z; from locations A, B, and C. The format is "loop sensor name": 8 bytes, "occupancy": 16 bytes, and "volume": 8 bytes."
>
> "XXX-4583" 50 3
> "XXX-4587" 43 2
> "XXX-3848" 48 2
> ...

More on the SDD format shortly...

What Kind of Actual Data Should be Made Self-Describing?

Any kind of data can be delivered as self-describing. But if you have an unchanging (static) set of data, such as a set of reports, or a very slowly changing set of data, such as the King County Metro Transit schedule data that is updated only three times a year, it may not be worth encoding and delivering this data using the SDD approach.

It may be that the best kind of data to be delivered in SDD format is real-time (or near-real time) data streams such as the WSDOT traffic loop sensor data or the King County Metro Transit bus location data. The data elements of these streams are constantly changing in content (remember loop sensor data is updated every 20 seconds), and they have a rich set of descriptive data that can be used by applications in providing traffic management information, traveler information, and so on.

About the Description

The real-time or actual data may be very simple in structure. Even if the actual data is simple, the descriptive data component of an SDD stream may be quite extensive and complex. This complexity is a confusing aspect of SDD. Elements of a simple data stream may have a lot of hidden or implicit meaning (We'll see an example of this below.). If users do not know this, data can be misinterpreted or misused. SDD tries to make more of this implicit meaning available through the descriptive data. Because of this potential complexity, descriptive data is encoded as a**database**, using database structures and language. This was designed so that applications could use relational database technology to manage large and complex data descriptions, as opposed to having to derive ad-hoc management schemes. The database is sent in two parts: a specification for the structure (or "schema") and the contents. So, changing the English prose we used above around a little bit, the stream would more closely look like:

> "Create one data base table that contains descriptions of the traffic loop sensors. Include a field for loop_id (16 bytes), a cabinet_id (7 bytes), etc. Create another database table that contains descriptions of the loop data. Include a field for sensor_id (15 bytes), volume (int), occupancy (int), etc...."
>
> "Put the following data in the Sensor table:
> "XXX-3848", "YY1", ...
> "XXX-4959", "YY2", ...
> Put the following data in the Loop table:
> ...
> ..."
>
> "XXX-4583" 50 3
> "XXX-4587" 43 2
> "XXX-3848" 48 2
> ...

This gives us a specification of the data base tables that describes the traffic loop sensors. This is followed by the contents to insert into those tables. Finally, the data stream is sent.

The descriptive data, that the SDD development team calls a **data dictionary**, is not actually encoded in English, but in a database language called SQL-92. SQL-92 is a standard database language that will allow developers to pass these commands directly to a database and have it understand and perform the appropriate actions (hopefully, create the database and store the values). In SQL-92, our English prose above looks more like:

```
CREATE SCHEMA    <<<this is the structure or schema definition>>>

CREATE TABLE SENSORS
(LOOP_ID CHAR (16),
CABINET_ID CHAR (7),
...
)

CREATE TABLE LOOP_DATA
(SENSOR_ID CHAR (15),
VOLUME SMALLINT,
OCCUPANCY SMALLINT,
...
)

TABLE SENSORS
COLUMN (LOOP_ID, CABINET_ID...)  <<<this fills the tables we created >>>
"XXX-3848", "YY1"...
"XXX-4959", "YY2"...

TABLE LOOP_DATA
...

"XXX-4583"  50  3    <<<then finally the data...>>>
"XXX-4587"  43  2
"XXX-3848"  48  2
...
```

This specification is the same as the previous one, only written using SQL-92 language[1]. The CREATE instructions would allow a database to create the specified tables. The TABLE specification with its COLUMN dependents allows the database to fill the tables with the included data. Finally, the data stream trails as before.

1 Actually, the section that describes the contents of the data dictionary is written in a SQL-like language that the SDD development team calls their "Content Language." The Content Language specifications are translated into SQL-92 by the SDD software.

More Wrapping for SDD

The story of the SDD format doesn't end with SQL-92. SDD streams are packaged much like Christmas presents with many layers of wrapping paper. To be compliant with a computing standard known as **ASN.1**, the data dictionary and the actual data are wrapped using Basic Encoding Rules (**BER**). Why ASN.1? (Which, by the way, is pronounced "ay-ess-enn-dot-1.") It's much too long a story for this paper, but it is a standard the National Architecture and other ITS standards groups prefer. Using BER means give each of the three components of SDD a type, a length and a value. The SDD development team has created software that includes a BER encoder on the SDD transmission side, and a BER decoder on the SDD receiver side. The SDD development team has created software that does this automagically, so we really don't have to worry about it too much. But if anyone asks if SDD is ASN.1 compliant, we can say "Yes!"

The BER-encoded SDD streams are then also encoded into "ITS Frames." An ITS Frame is a package developed by the SDD development team to make the network transfer easier. There is an ITS "Framer" at the SDD transmission side and an ITS "Deframer" on the SDD receiver side. The SDD code performs this work automagically as well, so we really don't have to worry about it either. Below is a diagram of the flow of SDD from a source generating data to a sink that would use it.

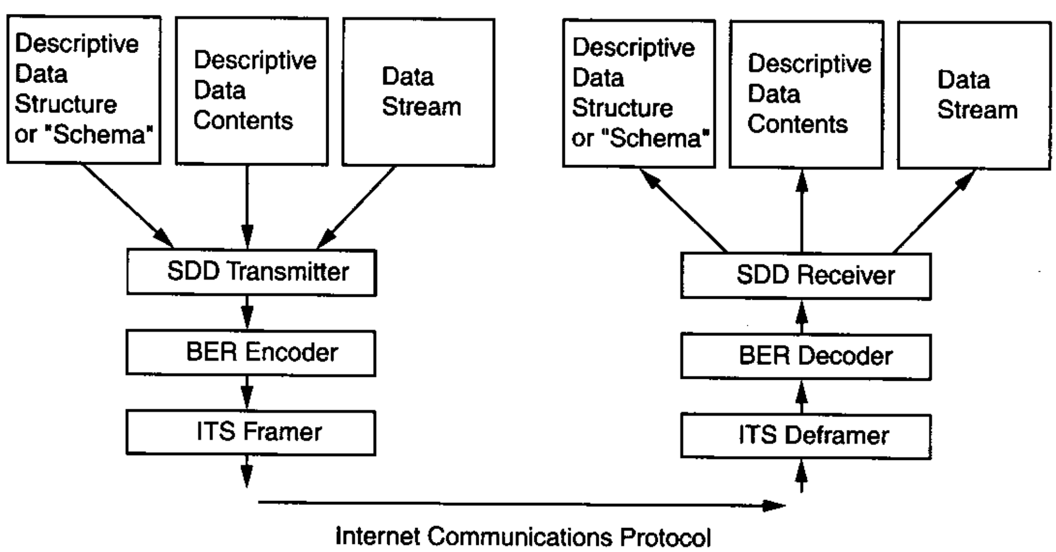


Figure 1 The flow of an SDD stream

What does an SDD stream look like that is BER-encoded and ITS-framed? You really don't want to see it. It's best to just think about SDD in terms of the original three components: the Data Dictionary Schema, the Data Dictionary Contents, and the actual Data.

What goes into a Data Dictionary?

Remember: *There are no rules as to what must be in an SDD data dictionary.*

We gave some examples above as to what might go into a data dictionary. They are not far from what has actually been encoded into one. The SDD development team has created extensive data dictionaries for WSDOT traffic loop sensors and an SDD stream is available with this data for ISPs and other interested parties to use. Let's look at that dictionary—it has 8 tables.

Here are the tables to be defined as part of the database:

COORDINATES - This table describes coordinate data types, such as "geodetic"; and their measurements, such as "longitude" and "latitude"; and their units of measure, such as "degrees".

MEASURES - This table provides additional data on coordinate data types, such as the fact that WSDOT uses NAD23 coordinate referencing, while UW uses NAD89.

STATION_FLAGS - This table provides flag values as to whether or not data is usable.

INCIDENT_DETECT - This table provides flag values as to whether an incident has occurred or not.

CABINETS - This table provides cabinet IDs, freeway names, text descriptions, and whether there is a ramp or not.

CABINET_LOCATION - This table provides the location of cabinets. It includes the cabinet ID, the coordinate type used (see above), and whether the data type is defined using the WSDOT or the UW methods, and the location.

LOOPS - This table describes the loop sensors. It contains the loop ID, the cabinet ID, whether or not it's metered, the road type, the direction of the traffic, the lane type, the lane number, and the sensor type code (a number).

ALG_DESCRIPT - This table provides a complete listing of Java code that will extract the loop sensor data!

As you can see, there is a lot of data here about the data. Heck, there's even the code to use it. If you didn't have access to this descriptive data, you would not automatically be aware of information such as locations and sensor types and so on. SDD makes that implicit data explicit and available. The user of SDD is not forced to use any of this descriptive information, but since it's being provided...why not?

When Do You Get a Data Dictionary and When Do You Just Get Data?

The data dictionary is sent upon connection to an SDD transmitter. An SDD transmission rule is that the data description remains valid until something changes in the data. So the data dictionary is sent once and is not sent again until there is a data change. When a data change does occur, a new data dictionary is sent through the stream. Again, if a data stream were being processed, and a change occurred in the data, the following might appear (going back to the English prose):

"XXX-4583" 50 3
"XXX-4587" 43 2
"XXX-3848" 48 2
"XXX-4589" 43 2
"XXX-3850" 48 2

"You'll now be getting traffic loop sensor data; that belong to these cabinets X, Y, Z; from locations A, B, and C. The format is 'loop sensor name': 8 bytes, 'occupancy': 16 bytes, 'volume': 8 bytes, and 'validity flag': 1 byte."

"XXX-4583" 50 3 1
"XXX-4587" 43 2 0
"XXX-3848" 48 2 1
...

The change that occurred was that a new data element was added to the stream called "validity flag." The reason for the new description is to allow users and smart applications that use SDD to handle the data change gracefully and perhaps even adapt to the change and continue functioning normally and without significant loss of service.

How Do You Know Whether or Not You Received a New Data Dictionary?

That is, without doing a lot of searches, data comparisons, and so on. It wasn't mentioned before, but the SDD stream also includes a time-stamp that is updated when the data dictionary changes. If an application sees a new time stamp, it knows something has changed. And yes, the SDD development team remembered to use four digits to represent the year portion of the time-stamp.

Software for SDD

We've mentioned an SDD transmitter and an SDD receiver, in terms almost like a television station transmitter and a home TV receiver, and this is just about how they work. Most ISPs are concerned with the "TV receiver" part of this setup, as they want to get traffic data. They are both pieces of software that projects can use to send and receive SDD. The receiver is the more mature piece of software at this writing, so we will review this.

The SDD receiver is a Java software program that can receive and understand the Data Dictionary and receive and extract the actual data from the SDD stream. The receiver has some quality checks in it that make sure the SQL-92 language is in the right format and the schema part is in sync with the contents part. The SDD receiver's primary job is to generate two files:

> An ASCII file that contains the SQL-92 code that can create the database for all of the descriptive data. This file can be given to a database system, such as ACCESS or SYBASE, to import the data.An ASCII file that contains the actual data.

So what do you do with this, if you're an ISP? Write software that can dump the Data Dictionary SQL-92 code into a data base, and write other software that can do something interesting with the data stream. That's the basic stuff. More advanced efforts will work with the newly formed database to provide very cool information to traffic managers or travelers.

How Do I Tap Into an SDD Data Stream?

SDD is delivered over the ITS INFORMATION BACKBONE (a.k.a. the "I2B", pronounced "eye-too-bee"). You plug into it using an Internet (TCP/IP) connection. Programmers know how to do these things. The address for the loop sensor data stream is "sdd.its.washington.edu" at port 9033. This provides the loops data on a 20-second cycle.

Want to see if this connection is for real (and you're not a programmer)? Open up Netscape on your PC and enter the URL: http://sdd.its.washington.edu: 9033/. Quickly, you will see a lot of data dictionary filling your screen. You won't be able to see the actual data decoded correctly, but you'll be able to tell if the port is active or not.

References

(1) Hall, Sydney R. "The star file: a new format for electronic data transfer and archiving," *J. Chem. Inf. Comput. Sci.*, vol. 31, no. 2, pp. 326-333, 1991.

(2) Wideman, Graham. "Streamlining experiment data manipulation with psycholog experiment data interchange format (pxdif),"*Behavior Research Methods, Instruments, and Computers*, vol. 23, no. 2, pp. 288-291, 1991.

(3) Allen, Frank H., John M. Barnard, Anthony P. F. Cook, and Sydney R. Hall, "The molecular information file (mif): core specifications of a new standard format for chemical data," *J. Chem. Inf. Comput. Sci.*, vol. 35, no. 3, pp. 412-427, 1995.

(4) NTIS. "Spatial data transfer standard: Fips 173-1," National Technical Information Service (NTIS) Computer Products Office, 5285 Port Royal Road, Springfield, VA 22161, 703-487-4650 Specify FIPSPUB 173-1, parts 1 through 4 when ordering., June 1994.

(5) Mark, Leo and Nick Roussopoulos. "Information interchange between self-describing databases," *Information Systems*, vol. 15, no. 4, pp. 393-400, 1990.

(6) Hall, Sydney R., Frank H. Allen, and I. David Brown. "The crystallographic information file (cif): a new standard archive file for crystallography,"*Acta Cryst.* vol. A47, pp. 655-685, 1991.

(7) Hall, Sydney R. and Anthony P. F. Cook. "Star dictionary definition language: initial specification," *J. Chem. Inf. Comput. Sci.*, vol. 35, no. 5, pp. 819-825, 1995.

(8) Johnson, John A. and Frederic C. Billingsley. "A standard method for creating self-defining data structures for information archive and transfer," *Digest of Papers. Tenth IEEE Symposium on Mass Storage Systems. Crisis in Mass Storage*, Washington, DC, USA., May 1990, IEEE, pp. 26-32, IEEE Comput. Soc. Press.

(9) Johnson, J. A. and F .C. Billingsley. "A standard method for creating self-defining data structures for information archive and transfer," *Digest of Papers. Tenth IEEE Symposium on Mass Storage Systems. Crisis in Mass Storage* (Cat. No. 90CH2844-9). K.D. Friedman and B.T. Olear, Eds., Monterey, CA, USA, May 1990, pp. 26-32, IEEE Comput. Soc. Press, Washington, DC, USA.

(10) Rockwell International and Lockheed Martin. "Theory of operations," in*Intelligent Transportation Systems. The National Architecture for ITS: A Framework for Integrated Transportation into the 21st Century.* FHWA Joint Program Office, January 1997, published on CD-ROM, uses Adobe Acrobat Reader, runs on Macintosh (68020 or greater or Power Mac) or Windows (386 or greater and Win

3.1, 95 or NT 3.51 or later), also available from www.itsa.org/public/archdocs/national.html or www.rockwell.com/itsarch/.

(11) Rockwell International and Lockheed Martin, "Physical architecture," in*Intelligent Transportation Systems. The National Architecture for ITS: A Framework for Integrated Transportation into the 21st Century.* FHWA Joint Program Office, January 1997, published on CD-ROM, uses Adobe Acrobat Reader, runs on Macintosh (68020 or greater or Power Mac) or Windows (386 or greater and Win 3.1, 95 or NT 3.51 or later), also available from www.itsa.org/public/archdocs/national.html or www.rockwell.com/itsarch/.

(12) NTCIP Steering Group. "National transportation communications for its protocol (ntcip) a family of protocols," 1996.

(13) Stallings, William. *SNMP SNMP-2 and RMON: Practical Network Management, 2nd Ed.*, Addison-Wesley Publishing Company, 1996.

(14) ISO. "Iso/iec 8824-1 information technology - abstract syntax notation one (asn.1): Specification of basic notation.," ISO/IEC Copyright Office, Case postale 56, CH-1211 Geneva 20, Switzerland, 1995.

(15) Dailey, D. J., M. P. Haselkorn, and D. Meyers. "A structured approach to developing real-time, distributed network applications for its deployment,"*ITS Journal*, vol. 3, no. 3, 1996.

(16) ANSI. "American national standard database language sql, ansi x3.135-1992,"American National Standards Institute, 1992.

(17) Codd, E.F. "A relational model of data for large shared data banks," *Communnications of the ACMe*, vol. 13, no. 6, June 1970.

(18) ISO. "Iso/iec 8825-1 information technology - asn.1 encoding rules: Specification of basic encoding rules (ber), canonical encoding rules (cer) and distinguished encoding rules (der)," ISO/IEC Copyright Office, Case postale 56, CH-1211 Geneva 20, Switzerland, 1995.