Title: A Self Describing Data Transfer Methodology for ITS Applications

Authors: D. J. Dailey, D. Meyers, N. Friedland

Transportation Research Board
78th Annual Meeting
January 10-14, 1999
Washington, D.C.
A Self Describing Data Transfer Methodology for ITS Applications

D.J. Dailey, D. Meyers, N. Friedland
University of Washington
Department of Electrical Engineering
Box 352500
Seattle, Washington 98195-2500
206-543-2493
fax 206-616-1787
dailey@its.washington.edu

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 (SDD) stream using: 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 (Abstract Syntax Notation One). Further, it documents a reference implementation of an SDD Receiver.

KEYWORDS

transportation data, ITS, standards, model deployments, SQL

ACKNOWLEDGMENT

This work was support in part by a series of contracts with the Federal Highway Administration and the Washington State Department of Transportation.

1. INTRODUCTION

The SmartTrek, Model Deployment Initiative, ITS Backbone is a set of protocols and paradigms designed to tie multi-agency geographically distributed ITS applications together. Applications, in this context, are a set of components that: extract ITS data from multiple source agencies, modify that data to create derived information products, distribute the data to Independent Service Providers (ISP’s), warehouse the data to create data mines, and use the data to create traveler information. The backbone is built around two key concepts: 1) a Component Architecture, described in “A Structured Approach to Developing Real-Time, Distributed Network Applications for ITS Deployment” (1) and 2) Self Describing Data (SDD) support, the topic of this paper.

This paper first presents an overview of the structure of SDD and discusses relevant standards. Two related standards, the National Transportation Communications for ITS Protocol (NTCIP) (2) and the Transportation Network Profile of the Spatial Data Transfer Standard (SDTS) (3), are used to provide a context for SDD. This paper describes the methodology used to create SDD and presents an example using traffic management data. It further details the actual implementation and use of SDD in the Smart Trek Model Deployment and describes an SDD receiver used to support several applications in both traffic management (North Seattle ATMS) and traveler information (Traffic TV, BusView, Transit Watch) applications. Finally, this paper describes the publicly available JAVA toolkit that implements SDD receivers and demonstration applications. The demonstration applications allow
anyone with an Internet connection to have both traffic and transit data from the Seattle area as well as providing as an automated database builder.

This paper provides a framework for an enabling technology to share traffic and traveler information between public agencies as well as providing a clearly defined open interface between public agencies and the private sector.

### 2. BACKGROUND

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 the sender and recipient of the data agree on a method for transfer. To date, most systems constructed for this purpose either lack generality or are limited to data transfers of a specific type (4, 5, 6) or they are so general and complex as to be very difficult to implement (3). The work presented in this paper is 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 ITS 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 (7). This 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 its location, the units of measurement, 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 itself. 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. There are a variety of proposed data transfer mechanisms that transfer the data and meaning (8, 9, 10, 5, 6). Many of the data sharing methods used to date have involved the construction of custom software that “understands” the meaning of the data to be transferred for each class of data transfer (6, 5). 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 very portable manner.

We present a new approach to solving the self-describing data 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. 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 (ISP’s) (See the ITS National Architecture study for more information on ISP’s (11, 12). We are aware of two other efforts to produce standards for communication of transportation related data. They are the National Transportation Communications for ITS Protocol (NTCIP) (2) and the Transportation Network Profile of the Spatial Data Transfer Standard (SDTS) (3). 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 NTCIP and SDTS.
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. 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 using 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 an extension of the Simple Network Management Protocol (SNMP). It extends the management information base hierarchical name-space defined (using ASN.1) for use with SNMP to include nodes devoted to the NTCIP. Groups of objects defined in this tree structured name-space are referred to as MIBs (Management Information Base) and represent the set of objects (in the object oriented design sense) that are needed to effect the desired control or information transfer operations. It 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 a control protocol which uses a globally agreed upon set of objects. To be effective, the control/management software that uses the get/set operations needs to understand how the software within the devices reacts to changes in the MIB. So there must be an a priori agreement on software functions as well as object definitions.

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 would then initiate or terminate an SDD transfer using the SNMP paradigms. The SDD transfer could be implemented either by viewing the MIB as a control that initiates an out-of-band data transfer as in (1) or the MIB could 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 and have those methods 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 (SDTS) that is designed for use with geographic vector data that has network topology. The SDTS allows transfers of spatial data that can be represented by vector objects that comprise a network or planar graph. The SDTS 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 SDTS 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 could be used to transfer time-varying data from a set of remote sensors by representing the time-invariant information about the sensors (location, sensor type, etc.) 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 are isolated into a separate table from the time-invariant information, it would be possible to send the module containing time-invariant data first and continue with the “open-ended” time-varying
module until no more time-varying data was desired at the receiving end. Though possible, transfer of large amounts of time-varying data using the SDTS 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 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.

3. DATA MODEL

In the work presented here, the data to be transferred is modeled as having 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 model, the Data Dictionary is comprised of two parts: 1) Dictionary Schema and 2) Dictionary Contents. These two parts provide the necessary description of the data to make it useful to a client and are described in the next sections.

3.1 Dictionary Schema

The first part of our Data Dictionary is the Dictionary Schema. This is meta-data that specifies 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 (15). We chose Entry Level SQL-92 because it is fully relational and the relational model can represent an arbitrary set of data (16). 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 that 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 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 the 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 use 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 is used to create and verify dictionary contents file.

The structure of the schema language is such that the maximum depth of the parse tree is four. Further, 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. A single schema node could be used as the root of the parse tree. We chose the 4-level tree implementation to make it easier to extend to higher levels of SQL-92 conformance (which allows multiple schemas and catalogs).

3.2 Dictionary Contents

The second part of the data dictionary is the Dictionary Contents. The dictionary contents is 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 degrees longitude, 47.21 degrees 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 are to 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 are to 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. On the sending end of a data transfer, the contents parser is used 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 is used to verify that the data in the contents file are compatible with the schema during SQL command generation.

3.3 Data Transfer

The overall architecture of our system for self-describing data transfer is shown in Figure 2. 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.

![Figure 2. An overview of the structure of our system for self-describing data transfers](image)

3.4 Transmitter

Figure 3 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. This parse tree is then used by the contents parser to verify that the contents complies with the schema. If both the schema and the contents are in compliance, they are serialized in the order schema, contents, and 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 (17). As a block of data is received, it is encoded and sent to all clients using the redistributor methodology detailed in (1). In our application, the data is a stream of bytes, and the information is extracted using algorithms specified in the Data
Dictionary. This mechanism allows for a very general transfer that is easy to encode, but which requires programming effort to be devoted to data extraction by the transfer recipient.

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) (17), with a type, length, and value.

We use the ASN.1 “Application” class with the “primitive” encoding, using tag numbers 1, 2, and 3 to 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:

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

The tag number needs to be unique across the family of protocols using these three data types. For example, if 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 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.

The SDD Receiver is a client application that converts from the data transfer stream format back to three data sets: schema, contents, and data. The structure of the receiver, as shown in Figure 4, is parallel to that of the transmitter in Figure 3. The BER decoder takes an SDD data stream as an input, BER decodes the data stream, and provides a decoded serial data stream that has the structure described in Figure 5. 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 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.

In our implementation, we have added an SQL generator. 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 result of an SDD transfer is a database with which an engineer can interact and a binary stream of dynamic sensor data. The practicality of this paradigm is demonstrated using an ITS example in the next section.
4. 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 this TMS traffic data is the focus for this ITS application of SDD. The data from the TMS consists of three parts: 1) sensor data which is binary data representing 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 comes 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, etc. make up the static component of the data dictionary contents.

Figure 6 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.

5. SDD APPLICATIONS

The Self-Describing Data (SDD) applications programming interface (API) of the ITS Backbone is depicted in Figure 7. The overall backbone design includes transmitters, operators, and receivers, stacked into three functional layers: Domain, SDD, and Frame, as shown in Figure 7. This paper describes the software presently available that
implements the receiver portion of the API (software and documents available at http://www.its.washington.edu/bbone). The receiver software included is represented schematically by the SDD 2.0.0 column to the right of the Receivers in Figure 7. The software is based in an object-oriented paradigm that maps into the three functional layers.

![Backbone API Layers](image)

**Figure 7. SDD Application Programming Interface**

The ItsFrameReceiver (see Figure 7) is a class that receives data from a network socket connection and produces ItsFrame events. It is initialized and connected to an SDD data source using a host name and data port number. Once the connection has been established, a number of ItsFrame objects are transmitted to the receiver. These are validated, serialized, and made available to registered classes via the itsFrameReceived callback method (see Figure 7), which takes an ItsFrame as input.

![SddAutoExtract Control Panel](image)

**Figure 8. Control panel for SddAutoExtractReceiver**

The SDD layer produces four individual types of data:

2. **Contents:** an object containing meta-data values to be inserted into one or more of the tables defined by the Schema.

3. **Extractor:** a compressed JAVA jar file containing a data factory class that can convert the raw binary into tabular objects defined by the data tables (those tables ending with “_DATA”) in the Schema.

4. **Data:** the raw binary data.

The domain level of the API allows users to access Extracted Data. Extracted Data is the result of expanding the raw binaries into tabular objects corresponding to the data tables defined in the Schema. These events are produced in the SddReceiver by taking the latest Extractor and applying it to each of the Data events in the incoming stream.

Two example applications are available. The first, SddAutoExtractReceiver, (the control panel is shown in Figure 8) allows the user to connect and get data from any Sdd data source. TMS Loop data is available at port 8411, and Automatic Vehicle Location (AVL) data is available at 8412. The user is required to set the host and port for the SDD data source, and this application produces output files containing the Schema, Contents, Extractor, Data, and ExtractedData. The user can specify which of these files to produce using check boxes.

The second application, SddDatabaseReceiver, loads Schema, Contents, and ExtractedData directly into a database, leveraging Java DataBase Connectivity (JDBC) capabilities. To run this application, a database engine with remote access capability and its corresponding JDBC driver is required. This application produces the control panel shown in Figure 9. The user specifies the SDD data source host and port, the JDBC class and URL descriptions, as well as the user name and password. The application contains several examples of class name and URL templates from Oracle and Sybase. A meta-data toggle informs the application to load the Schema and Contents into the database. When this toggle is set to off, only extracted data is loaded and the meta-data is ignored. The time required to load extracted data will vary with the number of inserts, the speed of the database host, and the speed of the connection from the client running the application to that host. If the ExtractedData events arrive faster than the host can insert them, “overflowing” events will be dropped.

![Figure 9. Control panel for SddDatabaseReceiver](image)

The SDD methodology described here is being deployed for ITS traveler-information applications in the Seattle metropolitan area in the form of the ITS Backbone for the SmartTrek Model Deployment. Information Service Providers use the SDD receiver to connect to the regional ITS backbone and obtain both the Traffic Management data just described or Transit Coach information.
REFERENCES


