

Release 2.0.0 Beta 5

Self Describing Data Java Support

1. Overview

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. The content and makeup of this data can change over time without the data provider notifying the users. For example, a traffic management center may have a set of inductance loops that provide data for Independent Service Providers (ISP) and other centers. The operators of this center may arbitrarily change the number, order, and/or type of sensors (e.g. substitute radar for loops) in the data they are transmitting. To be able to continuously use such data, changes need to propagate automatically to any downstream users.

To share data with time varying features requires that both the sender and the recipient of the data agree on a protocol to define the contents and meaning of the data. This protocol must allow the source of the data to communicate changes to the downstream users while preserving the meaning of the data. The software components of the package distributed here provide such a protocol in the form of a general transfer mechanism called *self-describing data* (SDD). An SDD transfer consists of a *Data Dictionary* followed by a continuous stream of raw sensor data (see Figure 1). The Data Dictionary leverages the power of conventional data description languages, specifically a subset of SQL92, to describe the meta-data properties of the subsequent sensor data stream. An SDD transfer ends either when a new data dictionary is received or when the transport protocol used to deliver the SDD is interrupted. The software in this package has been used to provide data feeds for a wide variety of ITS products and is applicable to a variety of data types and sensors.

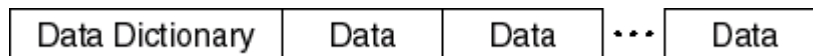


Figure 1: Self-Describing Data Transfer data stream.

2. Introduction

The Self-Describing Data (SDD) applications programming interface (API) of the ITS Backbone is depicted in Figure 2. The overall backbone design includes transmitters, operators, and receivers, stacked into three functional layers: Domain, SDD, and Frame, as shown in Figure 2. The software in this current release implements only the receiver portion of the API. The receiver software included is represented schematically by the SDD 2.0.0 column to the right of the Receivers in Figure 2. The software included is based in an object oriented paradigm, wherein the data types indicated in Table 1 are accessed using the callbacks, also identified in Table 1. In the callback model, classes register to be notified of events from event sources. When the event source produces an event, the callback methods in the registered classes are executed with the event as a parameter. The mapping of the various data types into the layered API model in Figure 2 is represented in Table 1.

Data Type	API Level	Callback	Class
ItsFrame	Frame	ItsFrameReceived	ItsFrameReceiver
Schema	SDD	SchemaReceived	SddReceiver
Contents	SDD	ContentReceived	SddReceiver
Extractor	SDD	ExtractorReceived	SddReceiver
Data (Raw)	SDD	DataReceived	SddReceiver
ExtractedData	Domain	ExtractedDataReceived	SddReceiver

Table 1. The API callbacks and data types.

The ItsFrame data type is a transport delivery construct used to pass serialized data between the transmitter and the receivers. These frames contain data encoded using the ASN.1 basic encoding rules (BER) and represent the various entities depicted in Figure 1. (For more information see both the SDD article <http://www.its.washington.edu/pubs/sdd.pdf> and report http://www.its.washington.edu/pubs/sdd_report.pdf)

The ItsFrameReceiver (see Figure 2) 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 2), which takes an ItsFrame as input.

The SDD layer produces four individual types of data:

- Schema: an object containing an SQL2 compliant data-definition language describing the data stream as a collection of tables.
- Contents: an object containing meta-data values to be inserted into one or more of the tables defined by the Schema.
- 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.
- Data: the raw binary data.

The SddReceiver receives ItsFrames via the itsFrameReceived callback from its internal ItsFrameReceiver. The receiver parses the ITS frame using the frame parser and expands the two BER packets contained in the frame. The first packet in an ITS frame contains a serial number - a 17 character timestamp with the format ‘yyymmddhhmmssmmm,’ (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 in the receiver to maintain state and relate meta-data to data. The second BER packet in the ItsFrame contains the SDD data type packet, which contains either meta-data: Schema, Contents and Extractor, or binary data.

Backbone API Layers

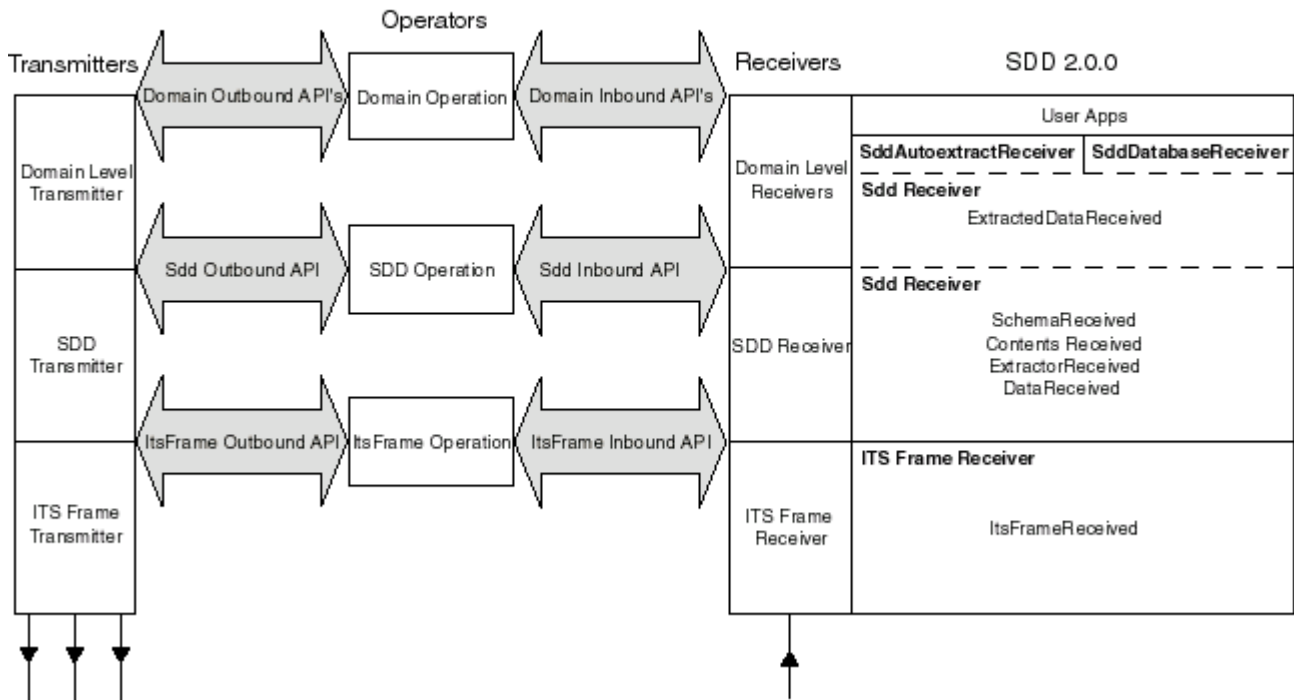


Figure 2: SDD Application Programming Interface

The serial number and the Sdd packet are packaged into an SDD event and passed back via one of the four SDD Receiver callbacks specified in Figure 2. The SDD Receiver monitors the SDD events and their associated serial numbers to maintain state. There are specific relationships between the serial numbers and the individual data types: (1) Schemas and Extractors must have the same serial number, (2) Contents and Data must have the same serial numbers, and (3) the Contents/Data serial numbers must be greater-than-or-equal-to the Schema/Extractor serial numbers.

To obtain the data, the user must register listeners for the various callbacks. Registered listeners will produce output in the sequence: Schema, Contents, Extractor events, followed by a stream of Data events, where every event in the data stream will have the same serial number as the current Contents until a new Contents arrives as part of a new Data Dictionary. See Table 1 for the Sdd event callbacks.

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. The domain level callback, in Figure 2, for Extracted Data is extractedDataReceived.

3. Installing Sdd2.0.0b5

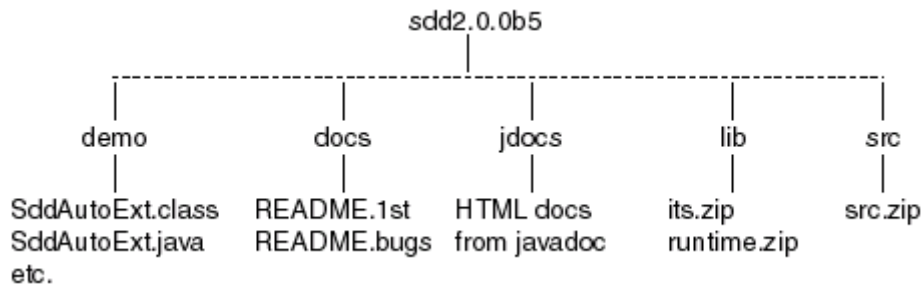
The Zip file for Beta of the ITS backbone can be found at

`ftp://ftp.its.washington.edu/pub/mdi/SDD/v2.0.0/sdd2.0.0b5.zip`

The URL above will work from a web browser; if you use anonymous FTP, ftp to `ftp.its.washington.edu` and get the file

`pub/mdi/SDD/v2.0.0/sdd2.0.0b5.zip`

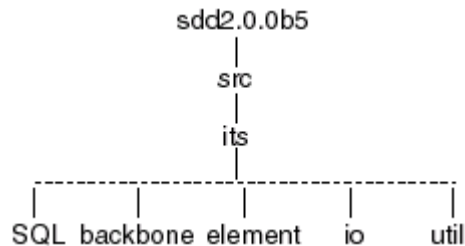
The directory structure in the zip file is similar to that of the JDK:



`sdd2.0.0b5/lib/its.zip` is an uncompressed Zip file suitable for placement in the user's CLASSPATH. It contains a classfile hierarchy rooted in "its" (e.g. `its/app/backbone`). **YOU DON'T NEED TO UNZIP THIS, YOU CAN USE IT AS IS!** These classfiles were produced from the source in `src.zip`.

`sdd2.0.0b5/lib/runtime.zip` is an uncompressed Zip file that contains Java classfiles that we are unable to reproduce from source.

`sdd2.0.0b5/src/src.zip` is a compressed zipfile containing source. If uncompressed the resulting tree will look like this:



(subdirectories are not shown here)

If the source in this directory is built, it will produce the classfiles in `its.zip`.

To test it:

- 1) Set your classpath to include these directories:

```
{ wherever you put sdd2.0.0b5 }\sdd2.0.0b5\lib\its.zip  
{ wherever you put sdd2.0.0b5 }\sdd2.0.0b5\lib\runtime.zip  
C:\jdk1.1.5\lib {replace jdk1.1.5 with correct version}
```

2) Cd to sdd2.0.0b5/demo and run one of the classfiles, e.g.

```
java SddAutoExtractReceiver
```

To build from source:

- 1) Set the classpath as above.
- 2) Change to the src directory and unzip src.zip.
- 3) Compile the Java code.

Due to the operation of the Java compiler's dependency checker, it is not necessary to compile all source files. The following two commands will compile enough of the source to run the demo application, SddAutoExtractReceiver, described in the next section:

```
(set classpath as above, this will build against  
its.lib)  
cd demo  
javac SddAutoExtractReceiver.java  
cd ../its/protocol/domain  
javac SensorData.java
```

The *src* directory contains src.zip, a zipped file with the Java source code. The *lib* directory contains classes.zip, an uncompressed zip file with the Java classes. The *docs* directory contains release and known bug documents. The *docs* directory contains the javadoc html pages for the release software. The *demo* directory contains two example applications.

4. Running the Examples

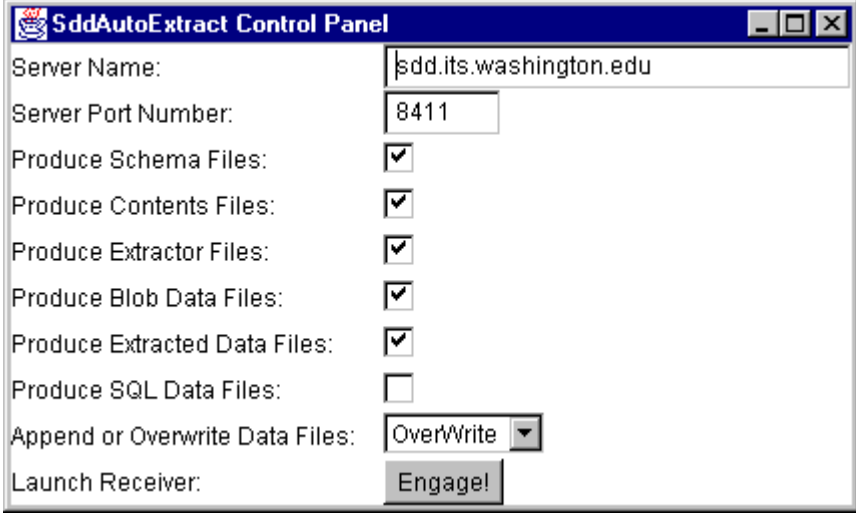
Two example applications have been provided for the user's benefit. The first, SddAutoExtractReceiver, allows the user to connect and get data from *any* Sdd data source. To run this application, set your classpath to:

```
.;  
<path to sdd2.0.0b5>/sdd2.0.0b5/lib/its.zip;  
<path to sdd2.0.0b5>/sdd2.0.0b5/lib/runtime.zip;  
<path to jdk1.1.5>/jdk1.1.5/lib/classes.zip
```

Next, go to its2.0.0b/demo and type:

```
java SddAutoExtractReceiver.
```

The following control panel will appear. At this time TMS Loop data is available at port 8411, and Automatic Vehicle Location (AVL) data is available at 8412.



The screenshot shows a Windows-style control panel window titled "SddAutoExtract Control Panel". It contains the following fields and controls:

- Server Name:
- Server Port Number:
- Produce Schema Files:
- Produce Contents Files:
- Produce Extractor Files:
- Produce Blob Data Files:
- Produce Extracted Data Files:
- Produce SQL Data Files:
- Append or Overwrite Data Files: (dropdown menu)
- Launch Receiver: (button)

The user is required to set the host and port for the SDD data source (8411=TMS, 8412=AVL). This application produces output files containing the Schema, Contents, Extractor, Data, and ExtractedData. The user can specify which of these files to produce. In all cases new files will be created every time a new serial number instance arrives. Data and ExtractedData files can be appended or overwritten. If append is specified for the binary Data, a log file is created, indicating the date, Binary Large Object (BLOB) size, and offset into the file, to allow the user to reconstruct the data parcels.

The second application, SddDatabaseReceiver, loads Schema, Contents, and ExtractedData directly into a database, leveraging Java's JDBC capabilities. To run this application, a database engine with remote access capability and its corresponding jdbc driver is required. The driver *must* be installed on the client machine. Consult the driver's documentation to obtain the necessary class name and URL.

To run this application, verify that your classpath includes sdd2.0.0b5, jdk1.1.5, the local directory, and the jdbc driver's classes as well. Type:

```
java SddDatabaseReceiver
```

The panel below should appear.

This application requires a running database server with a user and password on a specific instance of a database, and a jdbc driver for that database running on the client machine. From the jdbc driver documentation insert the jdbc class and url into the fields below. Type in the user name and password and select an Sdd host and port number. The checkbox indicates whether or not to insert metadata tables into the database.

DB vendor templates: **ORACLE**

Server Name: sdd.its.washington.edu

Server Port Number: 8411

DB JDBC driver class: oracle.jdbc.driver.OracleDriver

DB URL: jdbc:oracle:thin:@<myserver>:<myport>:<mysid>

User Name: scott

User Password: tiger

Insert Meta-Data:

Launch Receiver: Engage!

Specify the SDD data source host and port, as well as the jdbc class and URL descriptions. Then specify the user name and password. The application contains several examples of class name and URL templates from Oracle and Sybase. If your database/driver is from a different vendor, please consult your documentation to obtain the appropriate class and URL formats. A meta-data toggle informs the application whether to try and load the Schema and Contents into the database. When this toggle is set to off, only extracted data is loaded (the meta-data is ignored). The time it takes to load extracted data will vary with the number of required 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. Note also that the transmitted Schema does not contain state information. This will be critical against a time-varying data stream like TMS, where the BLOB contains readings from a variety of sensors, whose type and position within the BLOB is determined by the contents of the “LOOPS” table. Since the TMS schema cannot accommodate differing contents, the arrival of new sensor type and offset information will corrupt the existing instantiation. This makes SddDatabaseReceiver more of an example than a general application. **It is currently the responsibility of the parties receiving the data to handle these state transitions in their own data models!**

Both demo applications produce SddReceiverLog.txt files that document reception of various SDD events by the underlying SddReceiver.

5. New in SDD2.0.0

- Changes to the Event Model. The 1.0 event model contained separate callbacks for: SchemaEvent, Content(s)Event, DataEvent, ExtractorEvent, and SerialNumberEvent. These events, residing in its.backbone.sdd, all extended SddEvent. There was no explicit tie between the serial number event and its underlying SDD event - an important omission. 1.0 did not have any way of accessing exceptions thrown by SddReceiver. The 2.0 event model remedied this problem by modifying SddEvent to take a serial number as one of its constructor arguments. This SN can subsequently be accessed via the getSerialNumber method. These changes were propagated to the sub class events: SchemaEvent, ContentEvent, ExtractorEvent, DataEvent, and ExtractedDataEvent, all of which extend SddEvent in its.backbone.sdd. 2.0 dropped the ambiguous SerialNumberEvent, SerialNumberListener, and its callback method, serialNumberReceived. The other events retained their older constructors, but these were deprecated to alert the users to the change. 2.0 added the domain level ExtractedDataEvent and its listener, ExtractedDataListener. The 2.0 event model was also enhanced to include exception handling. Registered observers can now catch exceptions thrown in SddReceiver by evoking the event's getException method in the overloaded callback routine. If an exception is produced by the SddReceiver, getException will cause it to be thrown in the callback. Code that ran against the old 1.0 callbacks will need to be modified as follows:

1. Remove all references to serial number events, listeners, and callbacks.
2. To obtain serial number information, access the other events and evoke their getSerialNumber methods.

To utilize SddReceiver exceptions, see the example below for schema callback:

```
public void schemaReceived(SchemaEvent event) {
    try {
        event.getException(); // will throw exception if one was produced
        // other code
    } catch(Exception e) {
        // handle the exception
    }
}
```

- The SddReceiver has two new event callbacks: extractorReceived and extractedDataReceived. Both stem from a technology that leverages the JAVA ClassLoader capabilities. The former returns an extractor, a JAR formatted collection of class files containing a DataFactory, that allows the receiver to decode BLOB data into structured classes that emulate populated schema tables (see its.backbone.domain.DataFactory hyperlink). Since each data flow contains its own extractor, the receiver will be able to perform the decoding for *all* available Sdd data types. The latter callback returns those extracted structures, freeing users from having to perform the decoding themselves. We feel that this generic, domain independent capability will greatly enhance the ease-of-use of SDD.
- New applications (see Section 3). Our applications in the demo directory (especially SddAutoExtractReceiver) provide users with a simple tool that will allow them to connect to and configure incoming data from all available SDD data sources.

- A new its.SQL package, containing classes and methods that allow users to program and manipulate data at the schema level, including JDBC interactions against their DB host.
- Extensive, revised JAVADOC documentation describing the API and its packages.
- New, simplified directory organization that emulates a more conventional JAVA structure. The release provides all the class files in one easy-to-install zip file.