Final Research Report

Irregularly Sampled Transit Vehicles used as a Probe Vehicle Traffic Sensor

Daniel J. Dailey
Associate Professor
(Principal Investigator)

Chandrasekar Elango
Research Assistant

Department of Electrical Engineering
University of Washington
Seattle, Washington 98195

Prepared for:
Transportation Northwest (TransNow)
University of Washington
135 More Hall, Box 352700
Seattle, Washington 98195-2700

in cooperation with
U.S. Department of Transportation
Federal Highway Administration

March 1999

Contents

List of Figures

Abstract

Performance monitoring is an issue of growing concern both nationally and in Washington State. Travel-times and speeds have always been of interest to traveler-information researchers, but as a key measure in performance monitoring, this interest is now greater than ever. In this project, we use transit vehicles as probes and develop a framework for modeling the time series that arise from sampling transit vehicle locations as a function of time. These samples of vehicle location are obtained from the King County Metro Automatic Vehicle Location (AVL) system. An optimal filter method is developed that estimates speed as a function of space and time. In this work, an optimal solution for the state vector, containing the variables speed and position, is possible at each step using the Kalman filter result. This type of filter solution requires the creation of a model for the process; in this case, a relationship between location and time for the vehicles and the creation of a measurement model to account for measurement errors. Further, the use of such formalism depends upon the assumption that the deviations of the actual system from the idealized model are indeed normally distributed. The errors in the measurement model are taken directly from the specification documents for the operation of the AVL system. These performance specifications require relatively close tolerances on the errors in the actual vehicle location for a single sample (e.g., 100ft). Knowing the measurement properties, a linear relationship between position and time was postulated for the model of vehicle motion. This, in effect, suggests a constant velocity model for subsets of the travel path where the deviations from this model are identified as part of the randomness inherent in the process (e.g., stopping and starting are effectively noise). This is reasonable based on the infrequent and irregular manner in which the AVL system samples the position of the vehicles (e.g., 1-2 minutes between samples) relative to the actual motion of the transit vehicle. It was further postulated that a vehicle trip over the same route at the same time of day but on differing days is an ensemble realization of one process. With these postulates, the model was applied against data from both freeways and arterials to test the assumption that the deviation of the data from the model is

normally distributed. A Kolmogorov-Smirnov distribution membership test was used to validate the normality of the statistics of the residual differences between the data and the linear approximation. In most ranges of travel, the resulting probability of distribution membership (e.g., the probability of being normally distributed with the mean and variance predicted) is on the order of 0.9, indicating that the assumption of normally distributed errors is indeed a good one. [1]

Chapter 1

Background

Many researchers and public agencies would like access to immediate/real-time and historic travel-times and speeds in major urban corridors; however, corridor travel-time is difficult to estimate accurately, and inductance loops that estimate speed exist only in limited locations. There are several techniques that can be used to estimate travel-time and speed, each with its own strengths and limitations. The two techniques of interest here are:

- The use of inductance loop surveillance data. This technique directly estimates or derives speeds from other measures and then integrates these point measurements over space. While this is a readily available data source on freeways in cities such as Seattle, it requires assumptions about the validity of speed estimates as well as about traffic behavior over long stretches of freeway between the point measurements. Additionally, there are very few inductance loop data stations on arterial roads.

- The use of probe vehicles. Probe vehicle techniques are used to estimate the duration of actual trips for a single vehicle. This technology is typically a very limited source of data because continuously available travel-time and speed estimates from a variety of locations requires a large number of probe vehicles.

The Metro King County AVL-equipped transit fleet, which has on the order of 1000 vehicles operating simultaneously, can potentially provide the very large pool of probe vehicles necessary to make a probe vehicle strategy viable. The existing Metro AVL

system is based on using odometry and fixed routes to determine the position of a transit vehicle. This system was designed to be used for command and control by the transit agency dispatchers. Leveraging this system to estimate travel-time requires a clear understanding of the operation of the undocumented AVL system and the errors inherent in the data derived from the system. For example, the data available from the AVL system do not include vehicle position. Vehicle position must be calculated by combining information from the vehicle (e.g., distance traveled), information from the transit agency (e.g., the fixed routes traveled by each member of the fleet), and digital maps. Since those data about each vehicle are updated approximately every minute, multiple observations of a transit vehicle's location must be combined to create a statistically correct estimator of corridor speeds and travel-times. This becomes an optimal estimation problem in the face of varying errors in the position estimate. This problem is addressed here using optimal filter formalism with a system model to describe the time series produced by the AVL system as a vehicle travels a predetermined route.

The TransNow component of this project focuses on freeway corridor speeds and travel-time estimates. Freeway corridor travel-time is one of the principle tools used by local planning agencies, even though it is difficult to obtain accurate values. Further relating the observed travel behavior of transit coaches to the speed estimates that come from inductance loop technology remains a challenge. Metro King County transit vehicles are allowed to travel as fast as traffic without regard to schedule on the freeway (early arrivals are made up by layovers), and, as such, they are a potentially large fleet of unconstrained probe vehicles. However, these same vehicles are also unconstrained in their choice of travel lane. This lane uncertainty and the position estimate errors mentioned above make estimating corridor speeds and travel-times, as well as reconciling these estimates with other sensor technologies (e.g., loops and probe autos), a difficult problem.

In this report, we present the use of irregularly sampled positions of probe vehicles to estimate roadway traffic conditions. Probe vehicle techniques are sometimes used to estimate the duration of actual trips for a single vehicle and from this infer corridor travel conditions. This technology is typically a very limited source of data because

continuously available travel time and speed estimates from a variety of locations requires a large number of probe vehicles. However, the Metro King County automatic vehicle location (AVL) equipped transit fleet has on the order of 1000 vehicles operating simultaneously and can potentially provide a very large pool of probe vehicles. The existing Metro AVL system is based on using odometry and fixed routes to determine the position of a transit vehicle. For example, the data available from the AVL system does not include vehicle position. Vehicle position must be calculated by combining information from the vehicle (e.g., distance traveled), information from the transit agency (e.g., the fixed routes traveled by each member of the fleet), and digital maps. Since those data about each vehicle are updated irregularly, but approximately every minute, multiple observations of a transit vehicle's location must be combined to create a statistically correct estimator of corridor speeds and travel times. This becomes an optimal estimation problem in the face of varying errors in the position estimate.

Chapter 2

Data Collection, Reduction and Analysis

The ITS Backbone, created as part of the Seattle Smart Trek Model Deployment Initiative, is used to obtain the real-time transit vehicle information. The ITS Backbone is a set of protocols and paradigms designed to tie ITS applications together, and in this case, is used to extract information from King County Metro Transit's automatic vehicle location system. Representative Web pages that document the ideas and software that make up the ITS Backbone can be found in Appendix A. The Backbone pages provide a set of software that can be downloaded and used to obtain the AVL data as produced by Metro's AVL system. In addition, "pattern files" that represent the routes traveled by the transit vehicles were obtained from Metro Transit.

2.1 Data Collection

The static and dynamic transit data are both obtained using the Internet. The static data is comprised of pattern files and time point files that are ftp'd from Metro Transit. These files contain a representation of the spatial route over which an individual transit vehicle trip will travel.

2.1.1 Static Data

Three times per year Metro makes major changes to this data. These changes are as a result of schedule changes and happen in February, June, and September. Every other

week there are minor updates to this data. The data from Metro that is relevant to this project comes in the form of two ASCII files: pattern.dat and tpi.dat. The following is a description of the schema for each of these files.

tpi.dat : TPI stands for timepoint interval (or timepoint interchange). TPIs are a directional path between two timepoints. There is a starting timepoint, some points in between (called shape points), and an ending timepoint. The tpi.dat file, or table, contains all of Metro's tpis. A TPI is represented as a set of rows with the same tpi_id value. There is a unique tpi_id for each TPI. The first row of the TPI is the starting timepoint, and the last row is the ending timepoint. There is a from_tp column and a to_tp column for every row. The distance column is the distance from the start of the tpi. The sequence column is the index of that row of the tpi $(0, 1, 2, ...)$. The schema and data for tpi_id 39203904 is shown in Figure 2.1.

Name	**Type**	**Meaning**
tpi_id	int	unique identifier for a tpi
sequence	int	index of shape point
x_coord	double	state plane (WA North Zone) x coordinate
y_coord	double	state plane (WA North Zone) y coordinate
distance	double	distance from the start of the tpi
from_tp	int	starting timepoint
to_tp	int	ending timepoint

```
tpi.dat for tpi_id: 39203904
39203904|0|1281055.70492389|226968.291056725|0|3920|3904
39203904|1|1280796.00179829|226980.073604537|259.970271146443|3920|3904
39203904|2|1280509.11948201|226991.037850265|547.06203012694|3920|3904
39203904|3|1280251.85083928|226998.102262672|804.427646428396|3920|3904
39203904|4|1280220.91558418|226999.16020718|835.380986389668|3920|3904
39203904|5|1279938.28406872|227008.215634938|1118.15753121371|3920|3904
39203904|6|1279616.1494195|227011.364458175|1440.30756973895|3920|3904
39203904|7|1279295.27894678|227015.669121883|1761.20691593321|3920|3904
39203904|8|1278973.23946928|227020.125791517|2083.2772296434|3920|3904
39203904|9|1278651.48800974|227028.507450035|2405.13784219829|3920|3904
39203904|10|1278329.38755262|227037.868108177|2727.37428706146|3920|3904
39203904|11|1278010.43608909|227042.270762462|3046.35613521865|3920|3904
```

Figure 2.1: TPI schema and data.

pattern.dat: A pattern is the path that a route will follow. A pattern is an ordered set of TPI's. This file contains all of Metro's patterns for pay-service routes

but does not include dead-heads (non service trips). A pattern is represented as the set of rows with the same pattern_id value. There is a unique pattern_id for each pattern, and there is a pattern for each route. The definition of the schema for pattern.dat is shown in Figure 2.2.

Name	Type	Meaning
pattern_id	string	unique identifier for a pattern
pattern_seq	int	index of tpi in this pattern
tpi_length	double	length of the tpi in this row
tpi_id	long	unique identifier for tpi
from_tpname	string	starting timepoint name for this row's tpi
to_tpname	string	ending timepoint name for this row's tpi
to_tp	int	ending timepoint for this row's tpi

```
pattern.dat for route 2
00200504|0|5773.53|39133920|LK WASH E MADRONA DR|33 AV     E UNION ST|3913|3920
00200504|1|3046.35|39203904|33 AV     E UNION ST|23 AV     E UNION ST|3920|3904
```

Figure 2.2: Pattern file schema and data.

2.1.2 Dynamic Data

The second type of data used in this effort is the dynamic transit vehicle location data from Metro's Automatic Vehicle Location (AVL) System. To obtain this data, the ITS Backbone is used with freely available software located at:

`http://www.its.washington.edu/backbone`.

Included in this software are two clients named SddFlash and SddFilter to facilitate interactions with the data stream. These two applications are used to collect and analyze the transit data. SddFlash is a java client distributed with the its.app package. The client listens to the Sdd server at port 8412 and gives out a continuous stream of AVL data. SddFilter, a Perl program, parses the stream of AVL data and outputs AVL data from selected routes.

SddFlash: SddFlash is an ftp-like client that offers a command-line interface to inspect any SDD stream. Its output can be directed into a file or into any other

program for filtering or analysis. Files containing output from SddFlash and named with the suffix .csv can be opened directly by any program that can read text files. It is invoked by typing:

```
java its.app.SddFlash [-stream] hostname port [table]
```

This application prints a table from an SDD stream to standard output in a tabular format (comma separated, with column headers on the first line). The default behavior is to print the rows of data available from a single SDD frame and then terminate. This non-streaming behavior is useful for checking the status of a stream either from the command line or in other on-demand situations such as cgi scripts.

The table parameter is optional, and if omitted, a list of the tables available in the SDD stream will be printed. When the table parameter is specified, it can name a table whose data originates in either the Contents or Data frame of the SDD stream. More than one table can be specified, in which case the data for each will be printed, separated by a blank line.

If the "-*stream*" parameter is specified, the program does not terminate, but instead keeps listening for incoming data frames and printing the new rows of data it finds in each one. In this mode, the table or tables specified must originate in the SDD Data frame. If only a single table is specified, the column headers will only be printed once.

SddFilter: SddFilter is a program designed to examine the output of SddFlash and retain only those lines that match user-specified criteria. Other filter utilities such as grep (on Unix) and findstr (on Windows NT) can often be used just as effectively, but SddFilter allows the individual columns of data in each row to be examined and used in an arbitrary expression. An interpreter for the Perl language, version 5.0 or greater, is required to run SddFilter. The command to run sddfilter is:

```
perl sddfilter.pl criterion ...
```

Each criterion is a Perl expression involving the column names in the input SDD stream, which is read from standard input. Sddfilter expects input similar to that produced by SddFlash (comma-separated, with column names on the first line).

For example, to view the progress of all buses currently serving Metro Route 43, type:

```
java its.app.SddFlash stream sdd.its.washington.edu 8412 avl_data |
                perl SddFilter.pl svc_route == 43
```

Using these two programs, the AVL data is collected and filtered for further usage. The AVL data stream format is:

```
Vehicle_ID, Date_Time, Pattern_file, Last_Signpost,
    Distance_From_Signpost, Distance_from_start
```

2.2 Data Reduction

Several steps are taken to reorganize the data prior to analysis.

1. The data is sorted according to different patterns.
2. The pattern-sorted data is then sorted according to runs along the pattern on different days.
3. The date-sorted data is then sorted according to different runs along the pattern on the same day.
4. The trip-sorted data is then stripped to contain time in seconds and distance in feet. This was done to facilitate analysis in Matlab.

The data is then archived for future analysis.

2.3 Data Analysis

A linear relationship between position and time was postulated for the model of vehicle motion. This, in effect, suggests a constant velocity model for subsets of the travel path where the deviations from this model are identified as part of the randomness inherent in the process (e.g., stopping and starting are effectively noise). This is reasonable based on the fact that the AVL system samples the position of the vehicles infrequently (e.g.,

1-2 minutes between samples) relative to the actual motion of the transit vehicle. The linear relationship between time and space is quantified by estimating parameters for the linear model.

2.3.1 Assumptions

In this work it is also postulated that a vehicle trip over the same route at the same time of day but on differing days is an ensemble realization of the same process. The data containing the time and position information for a particular trip on different days along the same pattern file is amalgamated to estimate the parameters for the linear fit over the data. It is important to note that in this model there are two observed quantities, position and time. This implies that there are two dependant variables and no independent variable. In this case, regression techniques are not applicable and the parameters are estimated using the following approach:

1. The arterial data and freeway data are separated using information from the pattern files.
2. A linear regression is performed on the data to get the initial estimates for the parameters.
3. The data is then de-trended to compute the variance in position and time.
4. The variance, and the parameters from the linear regression, are used in the algorithm presented in the next section to estimate the parameters for the hypothesized linear model for the AVL data.

2.3.2 Maximum Likelihood Fit Algorithm for Dependant Variables

The general form for the model is

$$t = f(x; a, b). \tag{2.1}$$

We hypothesize that this relationship is linear, and the linear model is written

$$t = ax + b. \tag{2.2}$$

Unlike standard linear regression problems where there is a control variable (e.g., x) and an observed variable (e.g., t), this model has two observed variables (e.g., x, t) and no control variable.

The maximum likelihood methodology is chosen over the simpler linear regression based on two observations: 1) the observables used in estimating the parameters (a, b) are statistical quantities and 2) the statistics at each estimate are independent of the other estimates. These two conditions violate the assumptions necessary to use a simple regression technique. The following development provides a maximum likelihood methodology for estimating the parameters of our model in the context of our observables.

The deviation of a measurement from the linear model is written as

$$Error: \quad \epsilon_i = t_i - t(x_i; a, b). \tag{2.3}$$

We approximate these errors as being normally distributed so that the probability density function can be written as

$$f(\epsilon_1; x_i, t_i, a, b) = \frac{1}{2\pi\sigma_i} exp\left\{-\frac{1}{2}\left(\frac{t_i - y(x_i; a, b)}{\sigma_i}\right)^2\right\}, \tag{2.4}$$

where σ_i is the standard deviation of the errors ϵ_i. Now, if x_i and t_i are uncorrelated in the cluster of measurements for the ith site,

$$\sigma_i^2 = \sigma_{t_i}^2 + a^2\sigma_{x_i}^2 \tag{2.5}$$

and when we substitute 2.5 into equation 2.4 we get,

$$f(\epsilon_i) = \frac{1}{2\pi(\sigma_{t_i}^2 + a^2\sigma_{x_i}^2)^{1/2}} \times exp\left\{-(\frac{1}{2})\frac{(t_i - ax_i - b)^2}{\sigma_{t_i}^2 + a^2\sigma_{x_i}^2}\right\}. \tag{2.6}$$

A likelihood function for these statistics is written as

$$L(\epsilon; a, b) = \prod_{i=1}^{N} f(\epsilon_i; a, b). \tag{2.7}$$

Maximizing this likelihood function is equivalent to minimizing the negative of its log with respect to the parameters, and so the best estimate of the parameters satisfies

$$\frac{\partial}{\partial a}(-ln(L(\epsilon; a, b)) \quad = 0 \tag{2.8}$$

$$\frac{\partial}{\partial b}(-ln(L(\epsilon; a, b)) \quad = 0. \tag{2.9}$$

This results in coupled nonlinear equations. The intercept term (b) is written

$$b = \frac{\sum_{i=1}^{N} \frac{t_i - ax_i}{\sigma_{t_i}^2 + a^2\sigma_{x_i}^2}}{\sum_{i=1}^{N} \frac{1}{\sigma_{t_i}^2 + a^2\sigma_{x_i}^2}}. \tag{2.10}$$

The expression resulting from equation 2.8 is

$$\sum_{i=1}^{N} \frac{a\sigma_{x_i}}{A} = \sum_{i=1}^{N} \frac{2ABx_i + 2aB^2\sigma_{x_i}^2}{A^2}, \tag{2.11}$$

where

$$A \quad = \sigma_t^2 + a^2\sigma_{x_i}^2$$

$$B \quad = t_i - ax_i - b$$

does not result in a closed form expression for the coefficient a. Equations 2.10 and 2.11 are solved iteratively for the parameters a and b. This is done using a Newton's method.

An example of the fitting process for data from I-5 is shown in Figure 2.3, and an example result from SR-99 is shown in Figure 2.4.

2.3.3 Statistics and Distribution Membership Test

We use the Kolmogorov-Smirnov (KS) test to establish that the deviations of the AVL data from the linear model are normally distributed. The KS test statistic D is the maximum absolute difference between two cumulative distributions. A large value of the significance level of this statistic indicates that the two cumulative distributions are the same.

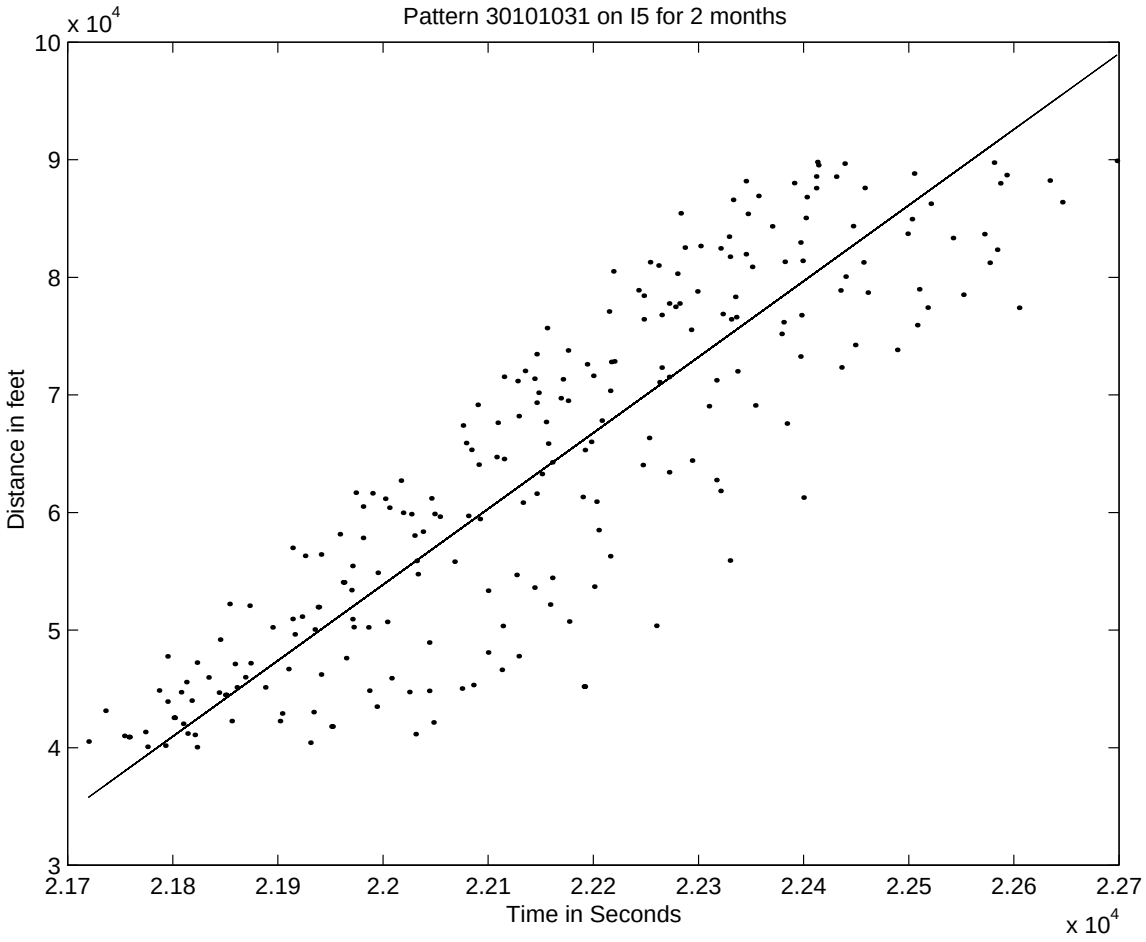


Figure 2.3: Transit position data for Interstate 5.

The error in the linear model is $\epsilon_i = t_i - f(x_i; a, b)$. Data points in 5000 foot intervals on the I-5 corridor are used, and the deviation of the data points from the linear model is computed. The mean and variance of the deviations are then computed, and the reference normal cumulative distribution ($\bar{P}(x)$) is constructed. The list of data points is converted to an unbiased estimator of the cumulative distribution ($S_N(x)$) which is the fraction of the data points to the left of a given value of x.

The K-S statistic D is

$$D = \max_{-a<x<b} |\bar{P}(x) - S_N(x)| \tag{2.12}$$

and is computed using the experimental results. The significance level of the statistic,

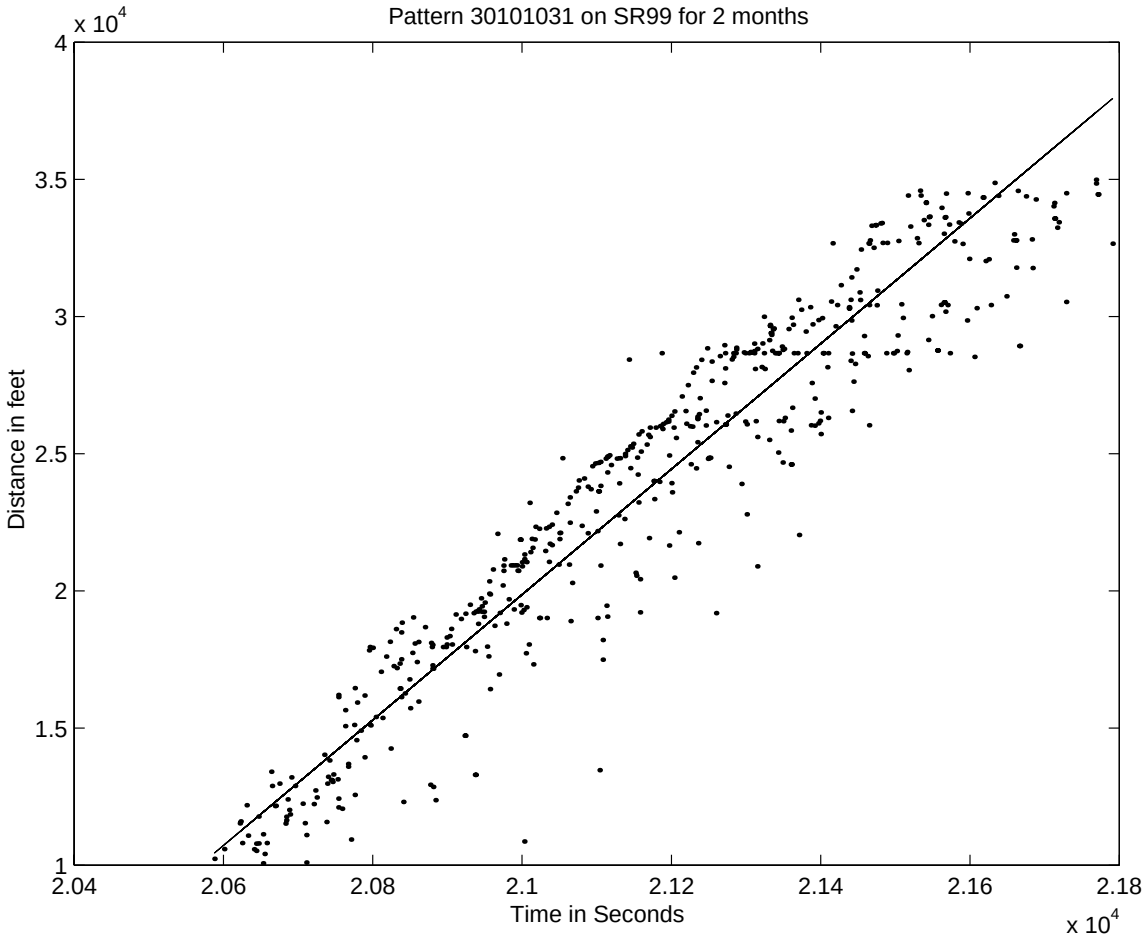


Figure 2.4: Transit position data for State Route 99.

given by [2], is

$$\text{Probability}(D > \text{observed}) = Q_{KS}\left([\sqrt{N} + 0.12 + 0.11/\sqrt{N}]D\right), \tag{2.13}$$

where N is the number of data points being considered and

$$Q_{KS}(\lambda) = 2\sum_{j=1}^{\infty}(-1)^{j-1}e^{-2j^2\lambda^2}. \tag{2.14}$$

Large values of the significance level of the probability indicate that the two distributions are the same. The same procedure is repeated for the arterial region where data points for every 500 feet are used.

Examples of the results of the normality test over the I-5 freeway portion of the path of the vehicle on service Route 301 are shown in Figure 2.5, and results for service Route 301 on the SR-99 arterial are shown in Figure 2.6.

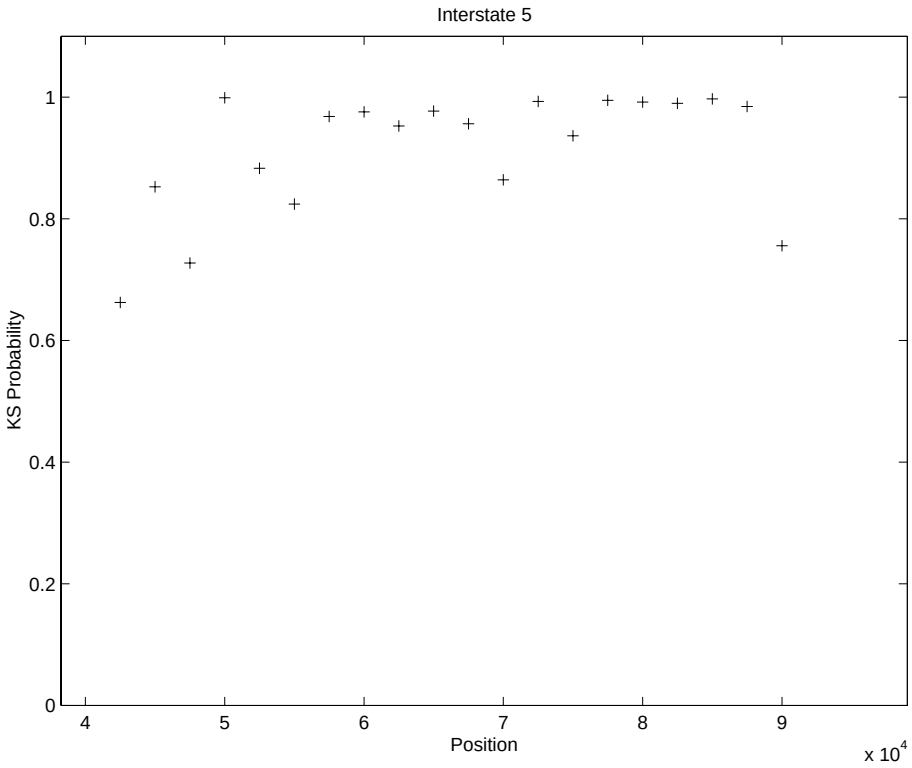


Figure 2.5: Transit position normality test for I-5.

2.3.4 Kalman filter

In this project, a framework for modeling the time series arising from the AVL system was developed and an optimal filter method was used to estimate speed as a function of space and time. The state vector for this model contains the position (x), the speed (s), the acceleration (a), and the jerk (j),

$$\mathbf{X} = \begin{bmatrix} x \\ v \\ a \\ j \end{bmatrix}. \tag{2.15}$$

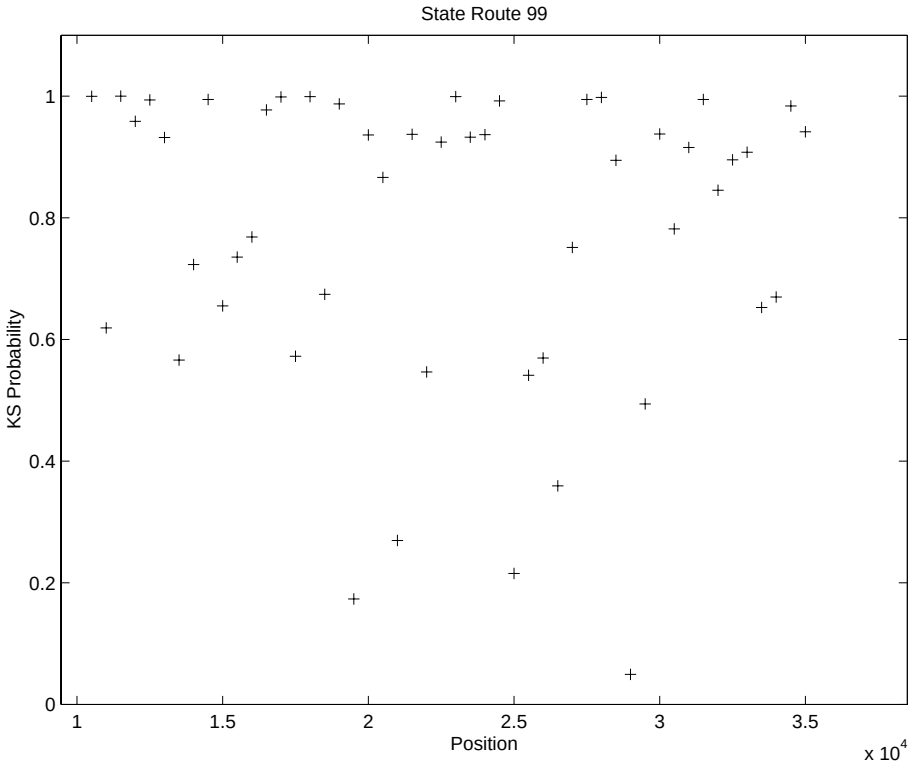


Figure 2.6: Transit position normality test for State Route 99.

The process model for this state vector is updated temporally, and these updates contain an additive normal random error (w),

$$\mathbf{X}_{k+1} = \mathbf{A}\mathbf{X}_k + w_k. \tag{2.16}$$

Further, the measurement vector ($\mathbf{Z}$) consists of the observed position (z) and time of observation (t)

$$\mathbf{Z} = \begin{bmatrix} \tau \\ z \end{bmatrix}. \tag{2.17}$$

The measurement model is

$$\mathbf{Z}_k = H_k\mathbf{X}_k + v_k, \tag{2.18}$$

where w_k represents the errors in the measurement process. With this problem description, an optimal solution for the state vector, containing the variables speed and position, is possible at each step using the Kalman filter result. The use of such a

formalism depends upon the assumption that the deviations of the actual system from the idealized model are indeed normally distributed.

In this project, the errors in the measurement model are taken directly from the specification documents for the operation of the AVL system, and a model for the update of the state vector was developed that resulted in normal deviations from the update model. The AVL system samples the position of the vehicles infrequently (e.g., 1-2 minutes between samples) relative to the actual motion of the transit vehicle. The AVL system itself has performance specifications that require relatively close tolerances on the errors in the actual vehicle location during a single sample (e.g., 100 ft). The system model for the update uses standard equations of motion,

$$\mathbf{A} = \begin{bmatrix} 1 & \Delta t_i & \frac{1}{2}\Delta t_i^2 & \frac{1}{6}\Delta t_i^3 \\ 0 & 1 & \Delta t_i & \frac{1}{2}\Delta t_i^2 \\ 0 & 0 & \frac{1}{1.05} & \Delta t_i \\ 0 & 0 & 0 & \frac{1}{1.1} \end{bmatrix} \tag{2.19}$$

and

$$\mathbf{H} = \begin{bmatrix} m & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}. \tag{2.20}$$

We have framed the estimation problem such that we can use the linear filter solution from reference [3],

$$\mathbf{P}_k^1 = \mathbf{A}\mathbf{P}_{k-1}\mathbf{A}^T + \mathbf{Q}_{k-1} \tag{2.21}$$

$$\mathbf{K}_k = \mathbf{P}_k^1 \hat{\mathbf{H}}_k^T \left[\hat{\mathbf{H}}_k \mathbf{P}_k^1 \hat{\mathbf{H}}_k^T + \mathbf{R}_k\right]^{-1} \tag{2.22}$$

$$\mathbf{P}_k = \mathbf{P}_k^1 - \mathbf{K}_k \hat{\mathbf{H}}_k \mathbf{P}_\mathbf{k}^\mathbf{1} \tag{2.23}$$

$$\mathbf{X}_k = \mathbf{A}\mathbf{X}_{k-1} + \mathbf{K}_k \left[\hat{\mathbf{Z}}_k - \hat{\mathbf{H}}_k \mathbf{A}\mathbf{X}_{k-1}\right], \tag{2.24}$$

where the noise contributions are

$$\mathbf{Q}_k = E\left\{\mathbf{w}_k \mathbf{w}_k^T\right\}, \qquad \mathbf{R}_k = E\left\{\mathbf{v}_k \mathbf{v}_k^T\right\}, \tag{2.25}$$

to update the state variables at each time step. This provides an algorithm to create the best single step estimate of the state variables that maximize the likelihood function given in equation (1) of reference [4].

The state variables are maximum likelihood estimates made using the Kalman filter with the observed data and postulated model. Example results for the state variable

estimates are shown in Figures 2.7 and 2.8. For the results from Interstate 5 (Figure 2.7), the constant speed approximation works rather well. The results for SR-99 (Figure 2.8) are less convincing. However, when the results from the normality tests in Figure 2.6 are examined, the areas where the normality assumption breaks down (low KS probability values) are the areas where the maximum likelihood estimates of the state variables seem most unrealistic. This phenomena, as well as comparison to speed estimates from inductance loop sensors, is the next step in the research.

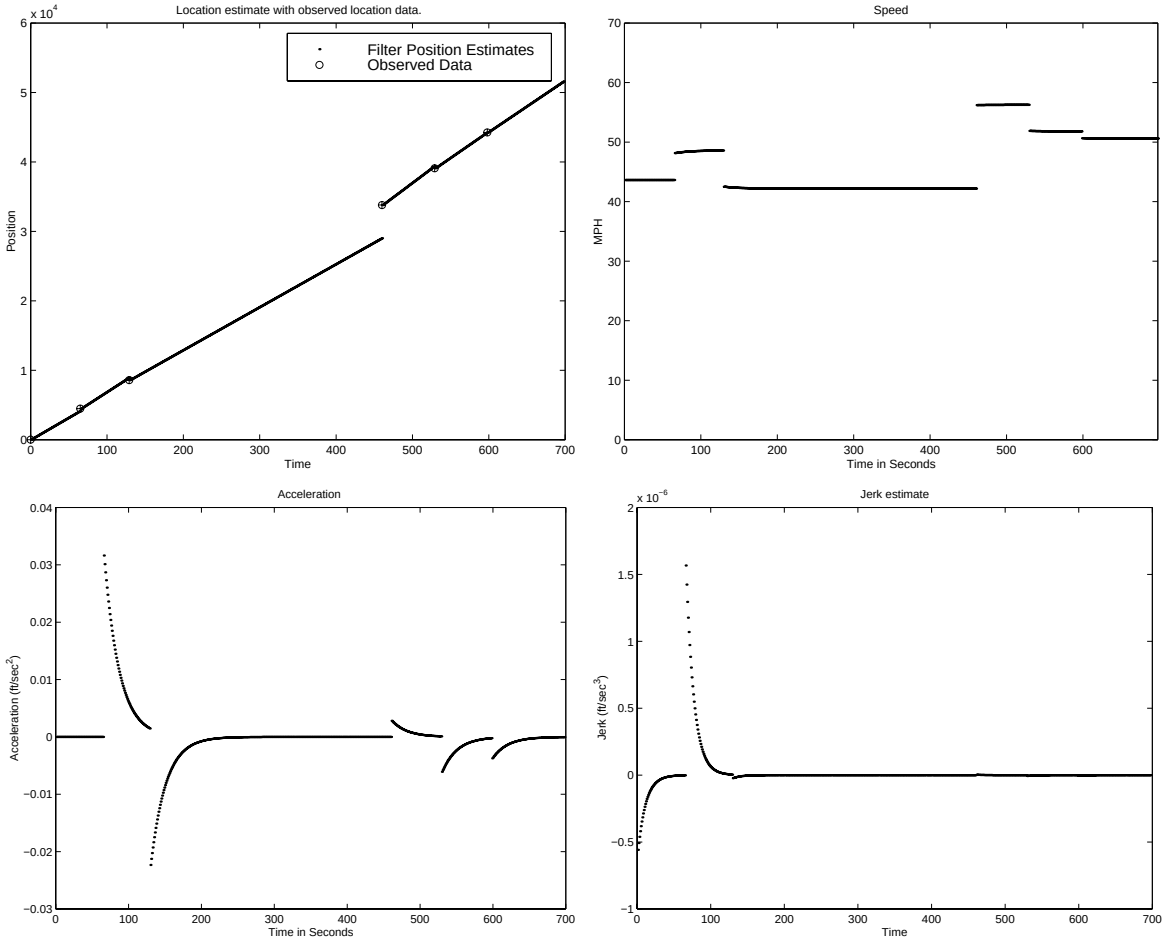


Figure 2.7: Position, speed, acceleration and jerk estimates for a vehicle on Route 301 traveling on I-5.

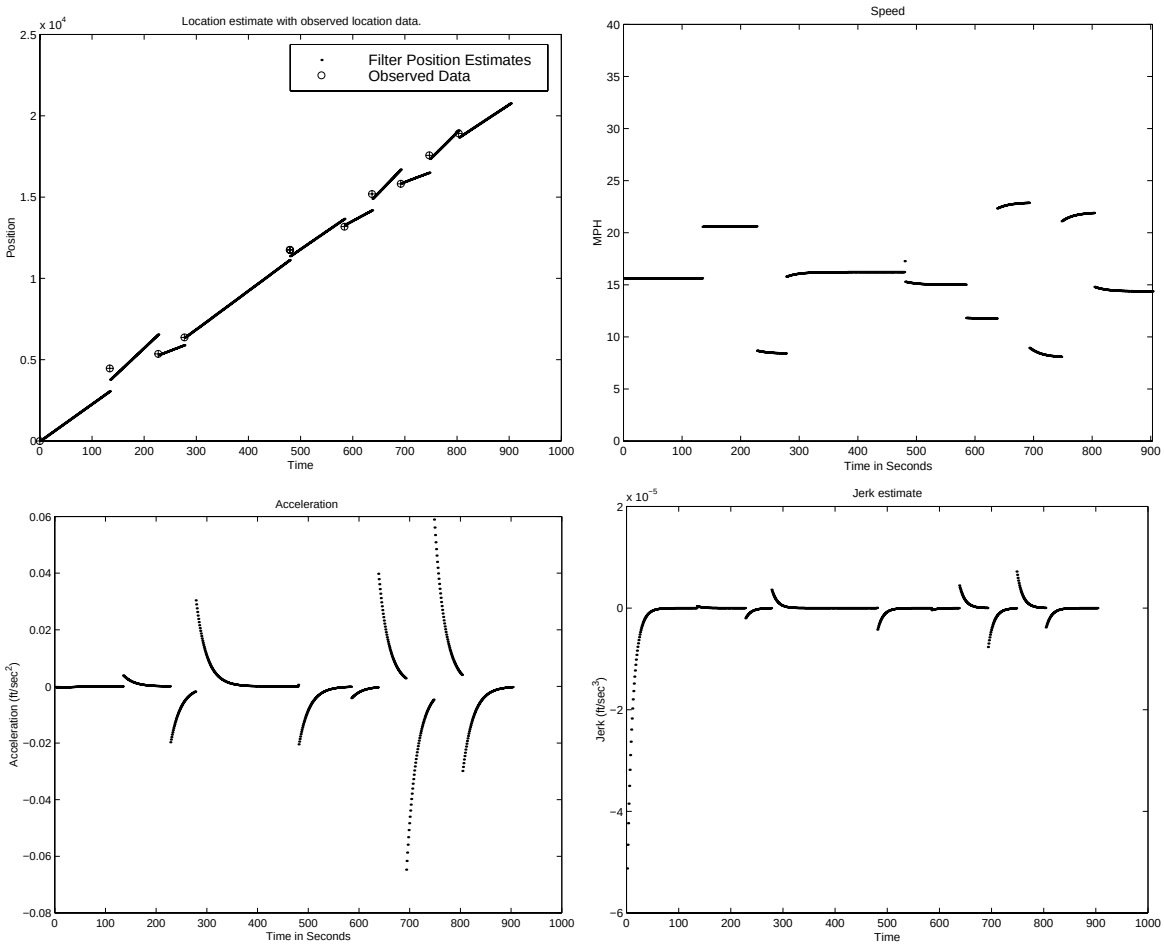


Figure 2.8: Position, speed, acceleration and jerk estimates for a vehicle on Route 301 traveling on SR-99.

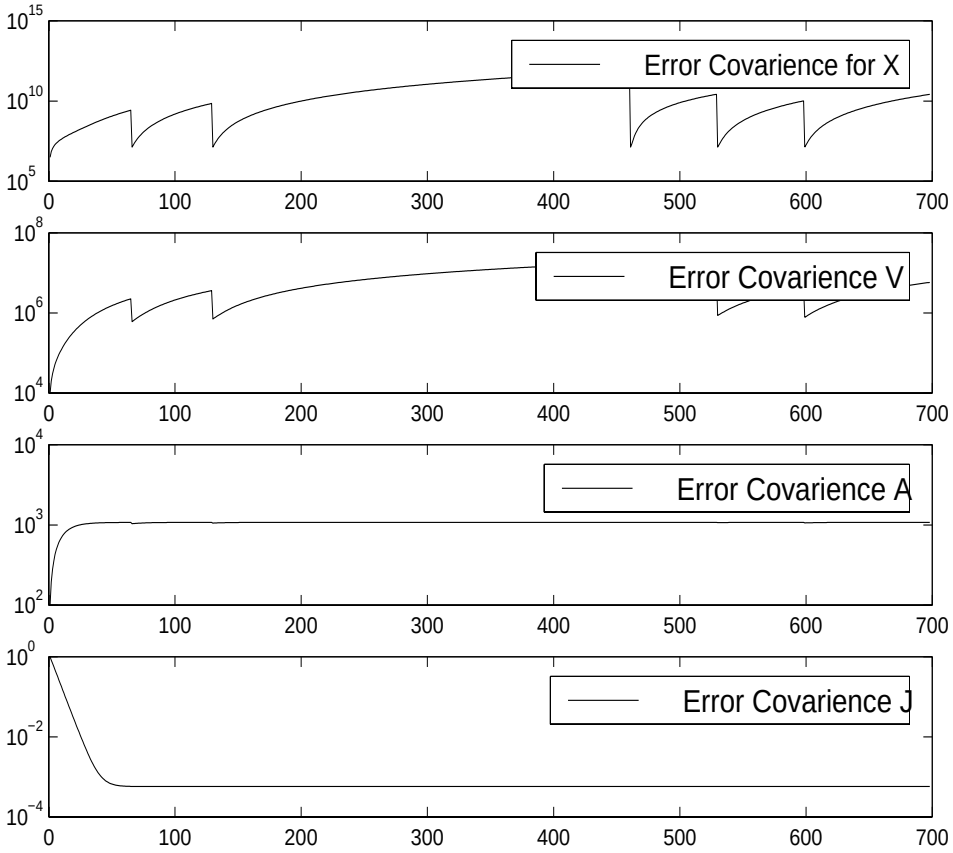


Figure 2.9: Error covariance for travel on I-5.

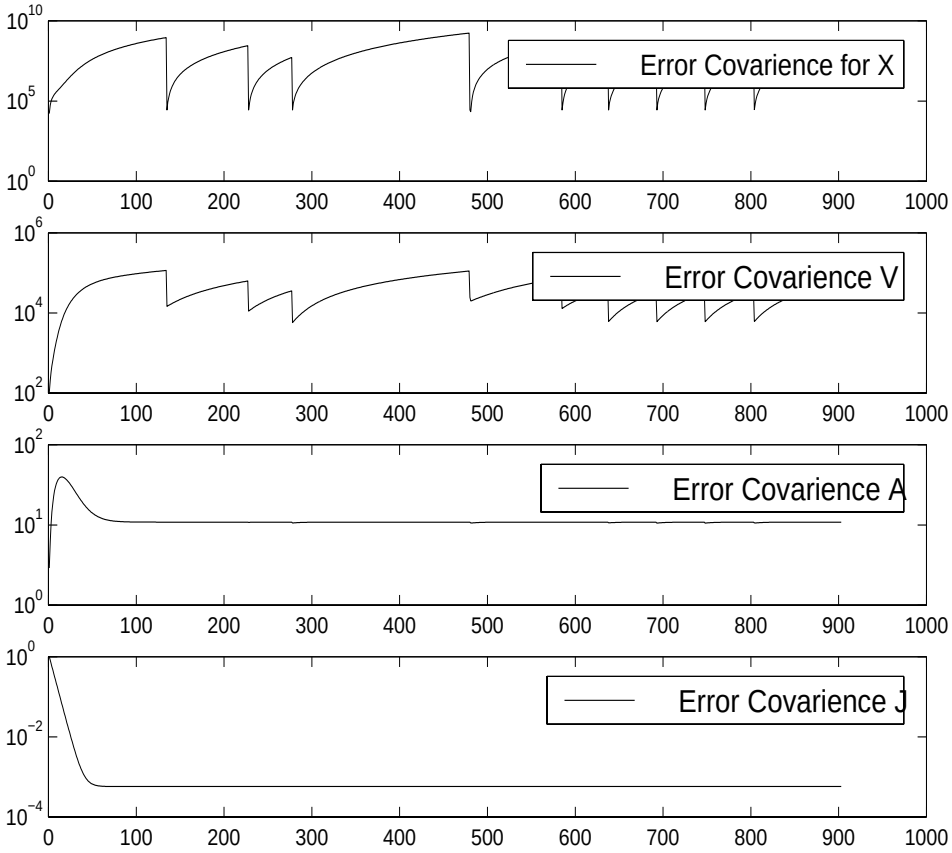


Figure 2.10: Error covariance for travel on SR-99.

Chapter 3

Conclusions

This work lays the groundwork for using King Country Transit's AVL-equipped fleet as probe vehicles. In this project, that uses transit vehicles as probes, a framework for modeling the time series arising from the AVL system is developed, and an optimal filter method is used to estimate speed as a function of space and time. In this work, an optimal solution for the state vector, containing the variables speed and position, is possible at each step using the Kalman filter result. This type of filter solution requires the creation of a model for the process; in this case, a relationship between location and time for the vehicles, and the creation of a measurement model to account for measurement errors. Further, the use of such formalism depends upon the assumption that the deviations of the actual system from the idealized model are indeed normally distributed. The errors in the measurement model are taken directly from the specification documents for the operation of the AVL system. These performance specifications require relatively close tolerances on the errors in the actual vehicle location for a single sample (e.g., 100ft). Knowing the measurement properties, a linear relationship between position and time was postulated for the model of vehicle motion. This, in effect, suggests a constant velocity model for subsets of the travel path where the deviations from this model are identified as part of the randomness inherent in the process (e.g., stopping and starting are effectively noise). This is reasonable based on the fact that the AVL system samples the position of the vehicles infrequently (e.g., 1-2 minutes between samples) relative to the actual motion of the transit vehicle. It is further postulated that a vehicle trip over the same route at the same time of day but on differing

days is an ensemble realization of the same process. With these postulates, the model is applied against data from both freeways and arterials to test the assumption that the deviation of the data from the model is normally distributed. A Kolmogorov-Smirnov distribution membership test was used to validate the normality of the statistics of the residual differences between the data and the linear approximation. In most ranges of travel, the resulting probability of distribution membership (e.g., the probability of being normally distributed with the mean and variance predicted) is on the order of 0.9, indicating that the assumption of normally distributed errors is indeed a good one. [1] Finally, a set of estimates of location and speed estimates made using a Kalman filter are presented. In all, this project demonstrates that the AVL-equipped transit vehicles can become excellent probe vehicles when the work presented here is extended.

Bibliography

[1] Z. Wall and D.J. Dailey. An algorithm for predicting the arrival time of mass transit vehicles using automatic vehicle location data. In *Transportation Research Board 78th Annual Meeting*, January 1999. Paper No. 990870.

[2] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in C*. Cambridge, 2nd edition, 1988.

[3] S.M. Bozic. *Digital and Kalman Filtering*. Edward Arnold, 1984.

[4] B.M. Bell. The Itterated Kalman Smoother as a Gaus-Newton Method. *SIAM J. Optimization*, 4(3):626–636, 1994.

Appendix A

Self Describing Data Receiver

Release 2.0.0 Beta 6

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.

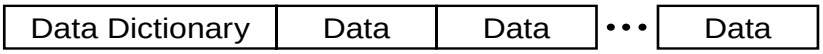


Figure 1: Self-Describing Data Transfer data stream

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 associated 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 '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 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.

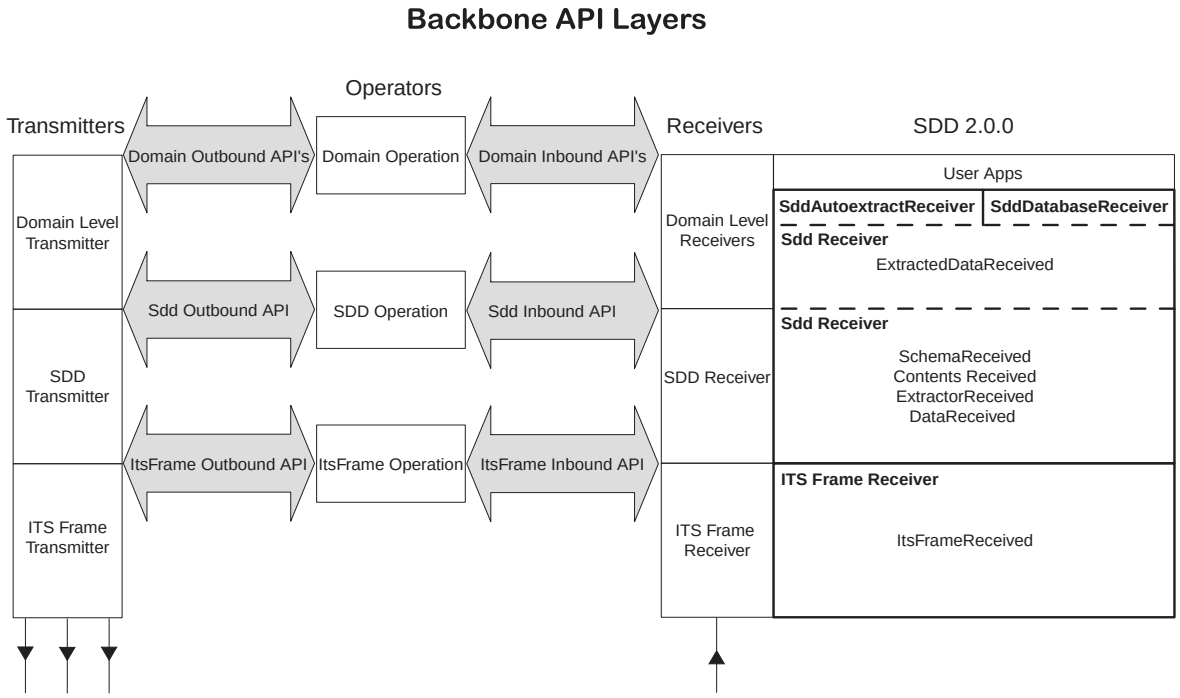


Figure 2: SDD Application Programming Interface

The serial number and the Sdd packet are packaged into an SDD event and passed back via one of the four SDD 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 SDD2.0.0b6

First, make sure you've installed the 1.1.6 release of the Java Development Kit. To test this, type "java -version." If this command does not return the string "java version 1.1.6," download the JDK at:

http://www.javasoft.com/products/jdk/1.1/index.html.

The Zip file for the beta release of the SDD Receiver can be found at:

ftp://ftp.its.washington.edu/pub/mdi/SDD/v2.0.0/sdd2.0.0b6.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/sdd2.0.0b6.zip

- Unzip this file in a local directory, for example, c:\sdd. You will find a "lib" folder that contains a file called its.zip. This file contains java classes and source code. It does not need to be unzipped to run the SDD Receiver; you can use it as is.

- Add c:\sdd\lib\its.zip to your classpath environment variable.

Windows users may wish to create batch files to set the classpath and run the example programs all at once. This helps prevent problems caused by editing your global classpath. A batch file to run the basic receiver would look like this:

```
REM Run the SDD Receiver
set classpath=c:\sdd\lib\its.zip
java its.app.SddAutoExtractReceiver
```

The other example program, SddDatabaseReceiver, requires a JDBC driver from your database vendor. Whatever package the vendor distributes must be on your classpath. So a batch file to run SddDatabaseReceiver would look like:

```
REM Run the SDD Database Receiver
set classpath=c:\sdd\lib\its.zip;c:\oracle\lib\classes111.zip
java its.app.SddDatabaseReceiver
```

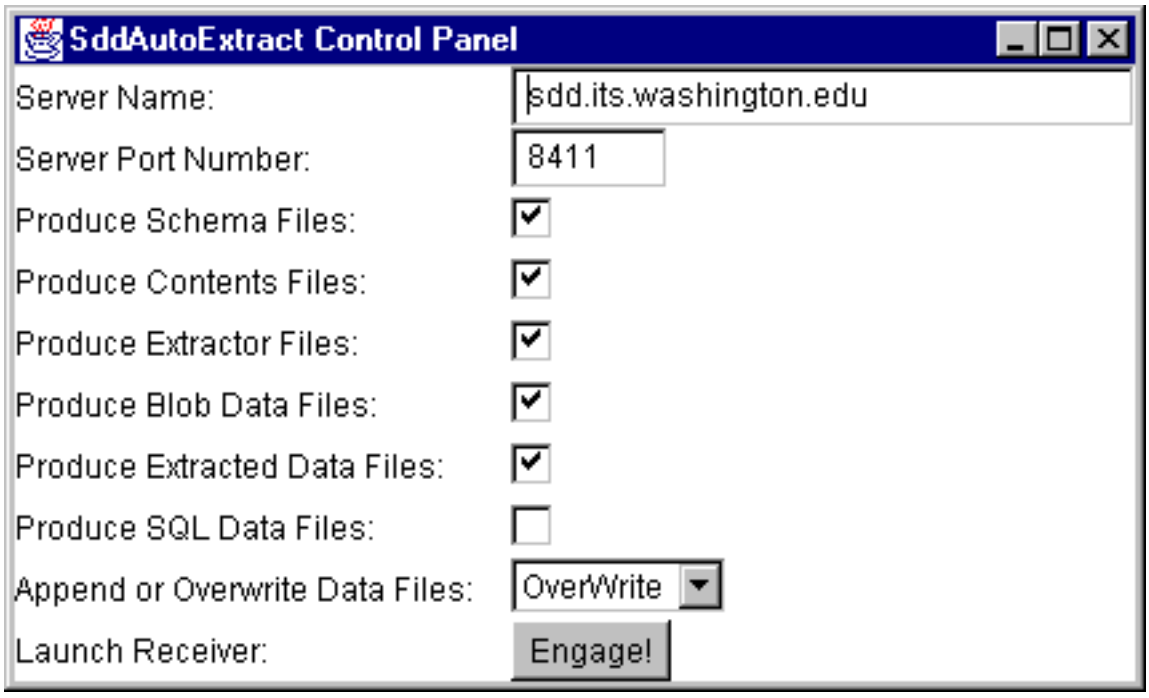
where “c:\oracle\lib\classes111.zip” is the name of the JDBC driver distributed by Oracle to access an Oracle database. If you are using a different type of database, contact the vendor for information about access through JDBC.

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, first make sure your classpath contains its.zip, as described in Section 3. Next, issue the command:

```
java its.app.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 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, indicat-

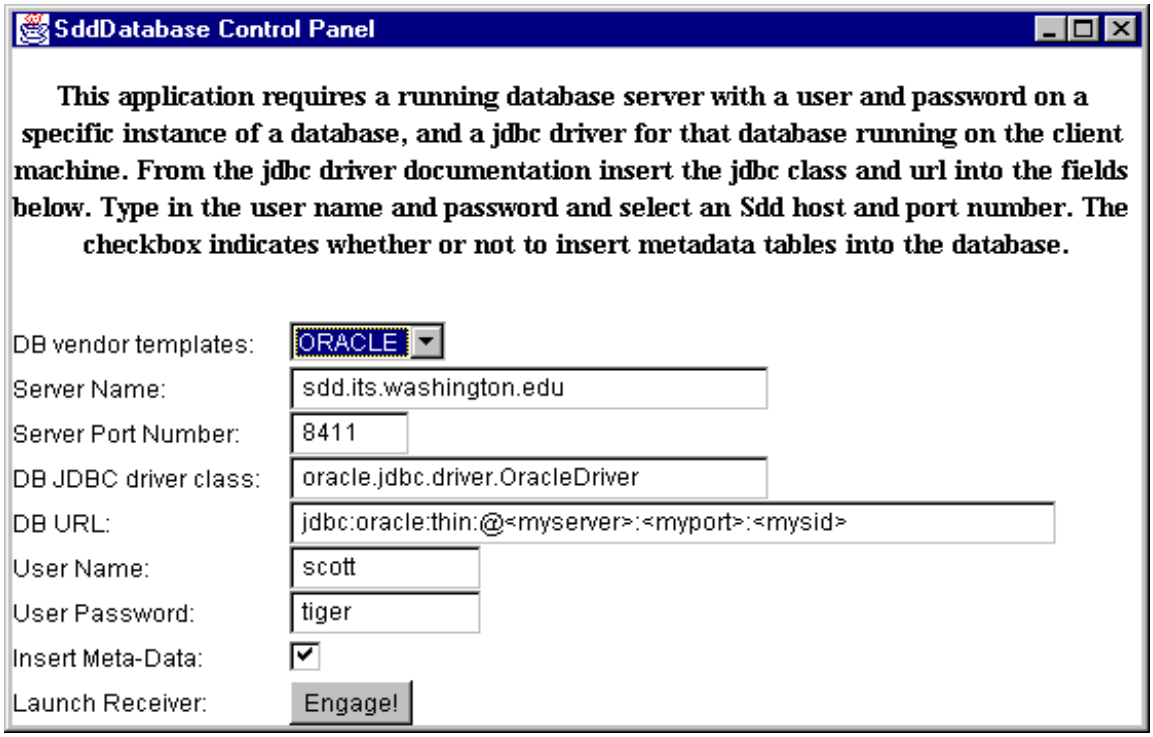
ing the date, Binary Large Object (BLOB) size, and offset into the file, to allow the user to reconstruct the data parcels.

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 its.zip, as described in Section 3, and the jdbc driver's classes as well. Type:

```
java its.app.SddDatabaseReceiver
```

The panel below should appear.



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. Recompiling the Source Code

Java programmers interested in recompiling the provided source code will need to unzip its.zip. The top-level “its” package contains the “SQL,” “backbone,” “element,” “io,” and “util” packages. All of these packages contain classes used by the receiver demos. To recompile the classes, delete all the class files and regenerate them with the java tools of your choice.

6. Release Notes

SDD Java API Release 2.0.0 beta 6

09/02/98

Overview

In this release, SDD parsing functionality is enabled in the receivers. The receiver will exit if it detects a poorly-formed schema or contents frame. This is an important step towards enabling creation of SDD transmitters for new data sources.

Bugs fixed

- In the receiver, when a new data frame arrives, it is passed to the extractor along with an “offset table” that describes where to find the data for each sensor. This offset table comes from the contents frame. In the case of traffic data, the offset inforamation is in the table called LOOPS. When a new contents frame arrives, the offset table should be regenerated, but in former releases it was not. So, the extracted data might not have been correct after the receiver had been running long enough to receive an updated contents frame.

- When the receiver performs automatic data extraction, error messages would appear if

table names in the schema or contents were not all upper case identifiers. Code in the SQL package has been updated to properly perform case-insensitive comparisons.

Other changes

- The two receiver implementations, SddAutoExtractReceiver and SddDatabaseReceiver, are now part of the its.app package. Once its.zip is on the classpath, the receivers are run using the command “java its.app.SddAutoExtractReceiver.”

- The its.zip file in the lib directory now contains the classes formerly distributed in runtime.zip. It also contains the source code formerly distributed in src.zip.

SDD Java API Release 2.0.0 beta 5

03/1/98

- 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 its.app package (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.