Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 18 (2013) 2036 – 2045

Procedia Computer Science

International Conference on Computational Science, ICCS 2013

Autonomous Data Error Detection and Recovery in Streaming Applications

Richard Klockowski, Shigeru Imai, Colin L Rice, Carlos A. Varela*

Computer Science Department, Rensselaer Polytechnic Institute, 110 Eighth Street, Troy, NY 12180, USA

Abstract

Detecting and recovering from errors in data streams is paramount to developing successful autonomous real-time streaming applications. In this paper, we devise a multi-modal data error detection and recovery architecture to enable automated recovery from data errors in streaming applications based on available redundancy. We formally define *error signatures* as a way to identify classes of abnormal conditions and *mode likelihood vectors* as a quantitative discriminator of data stream condition modes. Finally, we design an extension to our own declarative programming language, PILOTS, to include error correction code. We define performance metrics for our approach, and evaluate the impact of monitored data window size and mode likelihood change threshold on the accuracy and responsiveness of our data-driven multi-modal error detection and correction software. Tragic accidents—such as Air France's flight from Rio de Janeiro to Paris in June 2009 killing all people on board— can be prevented by implementing auto-pilot systems with an airspeed data stream error detection and correction algorithm following the fundamental principles illustrated in this work.

Keywords: redundant data error correction, spatio-temporal data streams, programming languages

1. Introduction

We present a software framework for developing resilient data driven applications and systems that act upon redundant spatio-temporal data streams. In this work we assume a spatio-temporal data streaming application model, where input streams associated to space and time get converted into output streams and error streams according to a mathematical description of the behavior of the application. Much like redundant bits in error correcting hardware, stream redundancies allow for dynamic detection and correction of *known* types of failures. Redundancy is a key aspect present in many spatio-temporal data streaming applications. However, unless it is effectively used by systems, autonomous recovery from error conditions is not possible. There are many complex ways in which a set of redundant input streams may fail. We propose a system towards automatically correcting known failures that can be detected in the source streams. We formalize *error signatures*, mathematical function patterns that enable autonomous systems to accurately detect when an erroneous condition exists in an input data stream. A multi-modal architecture uses these error signatures to switch each stream between different modes of operation. *Mode likelihood vectors* are computed

*Corresponding author
Email address: cvarela@cs.rpi.edu (Carlos A. Varela*)

Selection and peer review under responsibility of the organizers of the 2013 International Conference on Computational Science
doi:10.1016/j.procs.2013.05.373

in real-time by interpolating streamed data to a set of known error signatures. These vectors are used to determine the condition that input streams are exhibiting. When in a known error condition mode, the erroneous original data stream is automatically replaced by a data stream that is computed from the redundant (correct) data streams. The system dynamically adapts to errors in the data streams by switching modes, and it can resume normal behavior when input data is no longer categorized as being erroneous. We design an extension to PILOTS, a declarative programming language to not only compute error signatures from high-level specifications of spatio-temporal data streaming applications, but also to enable these applications to recover from known errors by using available redundancy in the data.

The Air France AF447 accident in June 2009 left 12 crew members and 216 passengers dead [1]. The reason for the crash was faulty sensor data that caused the automatic pilot to disengage, ultimately confusing the human pilots who were unable to take timely corrective actions. The pitot tubes of the airplane began to freeze which caused incorrect air speed readings, switching the plane from *normal law* to *alternate law*, and eventually causing the pilots to enter an unintended fatal stall. After a technical investigation by the Bureau d'Enquêtes et d'Analyses pour la Sécurité de l'Aviation Civile (BEA) it is clear that this error condition is detectable and can be corrected by an active redundant data-driven flight system. We argue that disasters like this are preventable by using an automatic pilot that implements the framework described in this paper: a multi-modal dynamic data-driven error correction software framework using error signatures and mode likelihood vectors.

2. Data Error Detection and Correction Architecture

Our contribution is an autonomous error correcting architecture for data streaming applications (depicted in Figure 1). The architecture was designed for applications with redundant input streams. For a set of input streams $D = \{d_1, d_2, ..., d_n\}$, the redundancy of the streams can be defined as the set of functions $R = \{r_1, r_2, ..., r_m\}$ where each r_i is a function $r_i(\hat{d}_1, \ldots, \hat{d}_k) = d_j$ for $j \in [1...n], k < n$, $\hat{d}_1, \ldots, \hat{d}_k \in D$ and $d_j \notin \{\hat{d}_1, \ldots, \hat{d}_k\}$. An *error function* associated to a particular input stream d_j may take the form $e_j = d_j - r_i(\hat{d}_1, \ldots, \hat{d}_k)$, where r_i is the redundancy function $r_i(\hat{d}_1, \ldots, \hat{d}_k) = d_j$. We define *error signatures* as the shape of these error functions for *previously known* error conditions. Additionally we formalize the *mode likelihood vector*, which is used to determine whether there is an error and whether or not it can be corrected. Data stream error correction is provided in the case that a redundancy within the other working streams is available. Especially in the case of spatio-temporal streams that use inherently redundant physical data such as those found in a flight system using sensor data, error signatures enable developing effective real-time error warning and correction systems. We contend that our proposed software framework can be useful to prevent tragedies such as the Air France plane crash. For this purpose we are developing PILOTS: a programming language for spatio-temporal data streaming applications [2]. PILOTS allows us to view heterogeneous data streams as homogeneous by declaratively selecting data according to geometric principles. In this paper, we describe an extension to the language design to include error correction code using the notion of error signatures. This software should prove very helpful for streaming application developers to enable them to create effective error correcting software.

2.1. Error Signatures

The purpose of an error signature is to be able to reason about which data stream may contain an error. A collection of error signatures, called an *error signature set*, is matched against the observed error which provides a means of error detection. We assume the existence of an *error function* which is simply a function of the input streams that captures the redundancy in the data streams. The *measured* error for an application is the value of the error function over a window of time. Each error signature corresponds to a particular type of failure in the input streams. The effectiveness of error signatures is highly dependent on the choice of error function. When there are no problems with the input streams, error functions typically evaluate to zero.

An error signature describes the behavior of the error function under particular operating conditions which we choose to call *modes*. An important distinction is made between *theoretical* error signatures, which correspond to known error modes, and *measured* error which is generated by looking at the raw input data. Theoretical error signatures are currently defined as a function of time which may contain constants $k_0, \ldots, k_n$ satisfying a set of constraints. In order to identify useful error signatures for a particular application, we currently employ an empirical method of simply running a simulation using data that exhibits a certain type of error and observe the results in the *measured*

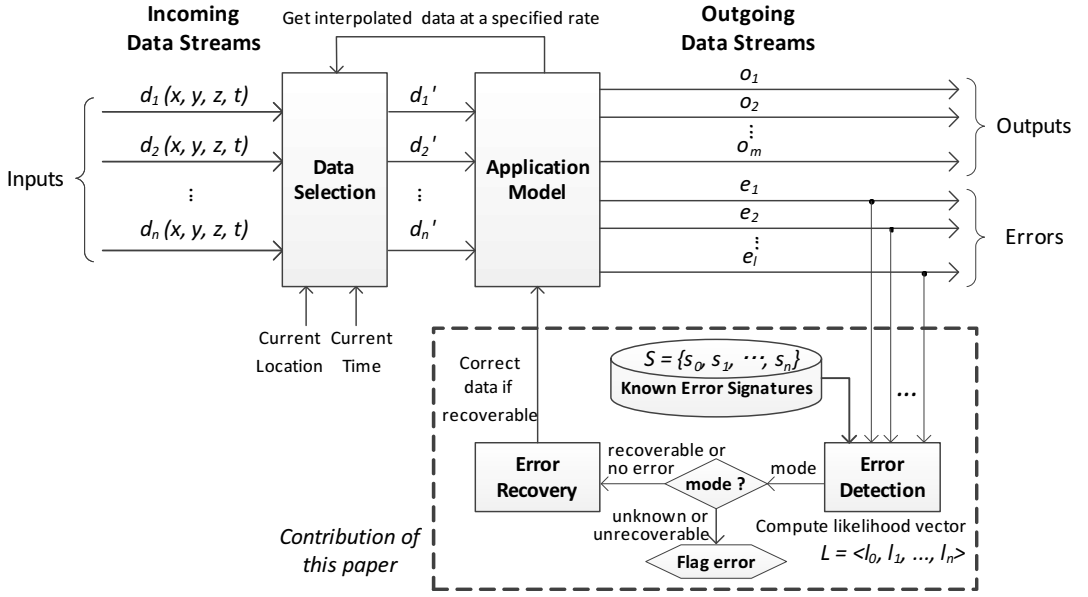

Figure 1: Data streaming architecture with error detection and correction

error. The error signature under *normal* conditions signifies that no errors have been detected. When all input streams are working properly the system assumes *normal* mode. Otherwise one of three modes is assumed: *unknown*, *recoverable*, or *unrecoverable*. If the system reaches *recoverable* mode, an error signature has been matched with the observed error and the appropriate redundancy is available to replace the stream producing the error. Thus for each error signature there exists a corresponding mode. If no redundancy is available the system switches to *unrecoverable* mode where a flag is raised (e.g., a red light bulb) denoting the type of error that was detected. In *unknown* error mode a similar type of flag is raised, but there is no known error signature that corresponds to the observed error. Only specific types of errors, those which have distinct error signatures and place the system into a recoverable mode, can be detected and corrected.

2.2. Error Detection

The measured error is compared to each of the theoretical error signatures in an attempt to find a strong match. Our current method for comparing error signatures is accomplished by formulating what we call the *mode likelihood vector*. Let $\{s_0, \ldots, s_n\}$ be the collection of known theoretical error signatures, where s_0 corresponds to the normal mode signature with no errors. We calculate the distance vector $\Delta(t) =< \delta_0(t), \ldots, \delta_n(t) >$ where $\delta_i(t)$ is the distance between the measured error $e(t)$ and $s_i(t)$. Specifically, $\delta_i(t) = \int_{t-\omega}^{t} |e(t) - s_i(t)|dt$ where $e(t)$ is the measured error and ω is the window size. The smaller the distance, the closer the raw data is to the theoretical signature. We formally define the mode likelihood vector to be $L(t) =< l_0(t), l_1(t), \ldots, l_n(t) >$ where each $l_i(t)$ is defined as:

$$l_i(t) = \begin{cases} 1, & \text{if } \delta_i(t) = 0 \\ \frac{\min\{\delta_0(t),\ldots,\delta_n(t)\}}{\delta_i(t)}, & \text{otherwise.} \end{cases}$$

Observe that for each $l_i \in L$ it follows that $0 \leqslant l_i \leqslant 1$, where l_i represents the ratio of the likelihood of signature s_i being matched with respect to the likelihood of the *best* signature. At each time stamp, the maximum two elements l_j and l_k of the mode likelihood vector, where $l_j > l_k$, are inspected in order to determine the error mode. Because of the way $L(t)$ is created, the maximum entry l_j will always be equal to 1. Given a threshold $\tau \in (0, 1)$ we check for one likely candidate that is sufficiently more likely than its successor by ensuring that $l_k \leqslant \tau$. Consequently, a *known* error mode is assumed. The correct mode is determined by choosing the error signature, and error mode, corresponding to l_j which is s_j. Each recoverable error mode uniquely determines the input streams that are erroneous. If $j = 0$ then

the system is in normal mode. If $l_k > \tau$ then, regardless of the value of j, *unknown error* mode is assumed and an error flag is raised. No corrective action can be taken because the measured error cannot be recognized, and the input data flows through the application uncorrected. A well-behaved set of error signatures will produce *nearly orthogonal* mode likelihood vectors, where one element is a one and the rest are close to zero. In sections 3 and 4 we study the impact of the choice of theoretical error signature sets on detection and correction results.

2.3. Error Recovery

If we assume that the system is in one of the known error modes (i.e., a match has been found for the measured error) then an attempt can be made at correcting the error. Recall that the *error function* is given and contains information about the redundancy between data streams. If an input stream d_j experiences an error and a redundancy r_i exists which can replace that stream, then the error is recoverable. After the error has been corrected, the original input streams will continue to be monitored to determine if the error has subsided and the system is able to reenter normal mode.

3. Twice: A Case Study

We explore how error signatures affect the values of mode likelihood vectors defined in Section 2 by using a very simple data streaming application called *Twice*.

3.1. A Simple Data Streaming Application

Twice is a simple data streaming application which takes two input data streams, a and b, where b is supposed to be twice as large as a, and outputs an error defined by $b - 2 * a$. Stream data for a and b are expected to increase by one for a and by two for b every second (i.e., $a(t) = t$ and $b(t) = 2 * t$), so the error is zero in the *normal* case; however, several modes of errors could happen depending on different types of failures as shown in Figure 2. Figure 2(a) shows *normal* mode, where most of the time the error remains zero, but there are several spikes due to transient fluctuation of the data input timing. Figure 2(b) suggests critical failure of a's data source. We will call this *a failure* mode. At around 50 seconds of the simulation time, the error starts growing linearly. This linear increase of the error explains that a remains a constant value whereas b continues increasing its value. Similarly, Figure 2(c) shows a situation where a correctly increases, but b fails to increase its value. Similarly, we will call this *b failure* mode. Figure 2(d) shows an example of an *out-of-sync* mode, where the error becomes consistently large at around 30 seconds of the simulation time. This is because a's input data stream becomes consistently one second behind b's input data stream.

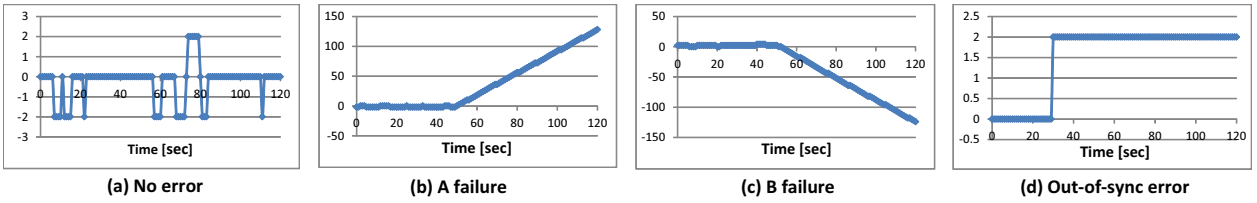

Figure 2: Known error patterns for twice example

3.2. Error Signatures for Twice

To correctly differentiate the four different modes of errors presented in Figure 2, we define error signatures for each mode. In the course of our case study, we evaluated three sets of error signatures as shown in Table 1. For the *no error*, *a failure*, and *b failure* modes, all error signatures are the same in these error signature sets. That is, $e = 0$ for no error, $e = 2t + k$ for *a failure*, and $e = -2t + k$ for *b failure*. Each error signature is designed to capture a characteristic pattern of error we see in the previous section. For example, an error signature for *a failure* is a linear function with a slope of 2 and a constant k, which resembles the increasing line starting at around 50 seconds of *a failure* shown in Figure 2(b). Differences among error signature sets are limited to the *out-of-sync failure* mode. Both *base* and *out-of-sync restricted* error signature sets have $e = k$ for the *out-of-sync mode*, but the *out-of-sync restricted* imposes a constraint on the value of k ($|k| > \tau_{oos}$). The τ_{oos} threshold is intended to prevent noise and small out-of-sync conditions from being categorized as abnormal.

Table 1: Error signature sets defined for *twice* example

Error signature set	Mode			
	No error	A failure	B failure	Out-of-sync
Base	$e = 0$	$e = 2t + k$	$e = -2t + k$	$e = k$, where $k \neq 0$
Out-of-sync restricted				$e = k$, where $\|k\| > \tau_{oos}$
Out-of-sync removed				none

3.3. Mode Estimation Study

Using the error signatures defined in Table 1, we estimate the operating modes for a 480 seconds sequence of measured error including mode change within sixty-second intervals as shown in Figure 3(a). Figure 4(a) shows the ground truth: the transition of modes that is used to generate the streams. In Figure 4, modes are mapped to 0 for *unknown*, 1 for *no error*, 2 for *a failure*, 3 for *b failure*, and 4 for *out-of-sync*. For each set of error signatures presented in the previous section, we first compute the likelihood of each mode, and then estimate mode likelihoods relative to the maximum likelihood which represents the minimum signature interpolation distance.

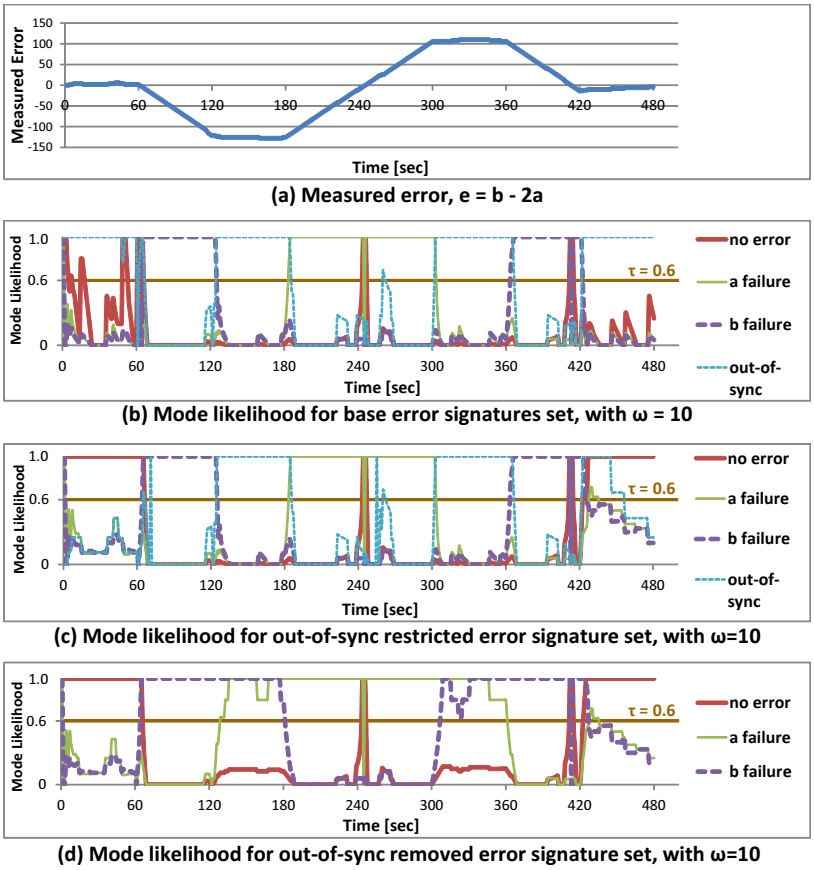

Figure 3: Measured error and mode likelihood results for twice example

The results of the mode likelihood and the estimated modes are shown in Figure 3(b)-(d) and 4(b)-(d) respectively. Figure 3(b) and 4(b) are results for the base error signatures, Figure 3(c) and 4(c) are results for the *out-of-sync restricted* error signatures, and Figure 3(d) and 4(d) are results for the *out-of-sync removed* error signatures.

- **Base:** Looking at the result of estimated mode in Figure 4(b), most of the first and last 60 seconds are recognized as the *out-of-sync mode*, where they are actually supposed to be in the *normal* mode. This incorrect estimation

occurs because the given error in Figure 3(a) contains noise so that the actual value of the error is not always exactly zero. Thus, the *out-of-sync* error signature fits better to the measured error than *no error* since the constant k in the *out-of-sync* error signature can be any value other than zero. This essentially illustrates an ill-defined error signature set: since *normal* and *out-of-sync* conditions are difficult to distinguish from each other, computed mode likelihood vectors are far from orthogonal.

- **Out-of-sync restricted:** The result of estimated mode in Figure 4(c) looks closer to the true mode in Figure 4(a) than the result of the *base* error signature set. The threshold τ_{oos} ($\tau_{oos} = 20$ for this experiment) is a constraint on the *out-of-sync* error signature that prevents it from matching the first and last 60 seconds, and correctly lets the *normal* signature match those periods instead.
- **Out-of-sync removed:** The result of estimated mode in Figure 4(d) does not match the *out-of-sync* mode at around 120-180 and 300-360 seconds since that mode does not exist, but it successfully goes into *unknown* error mode which are valid estimations when using this signatures set. Also, it succeeds to match *normal* mode at around 0-60 and 420-480 seconds most of the time.

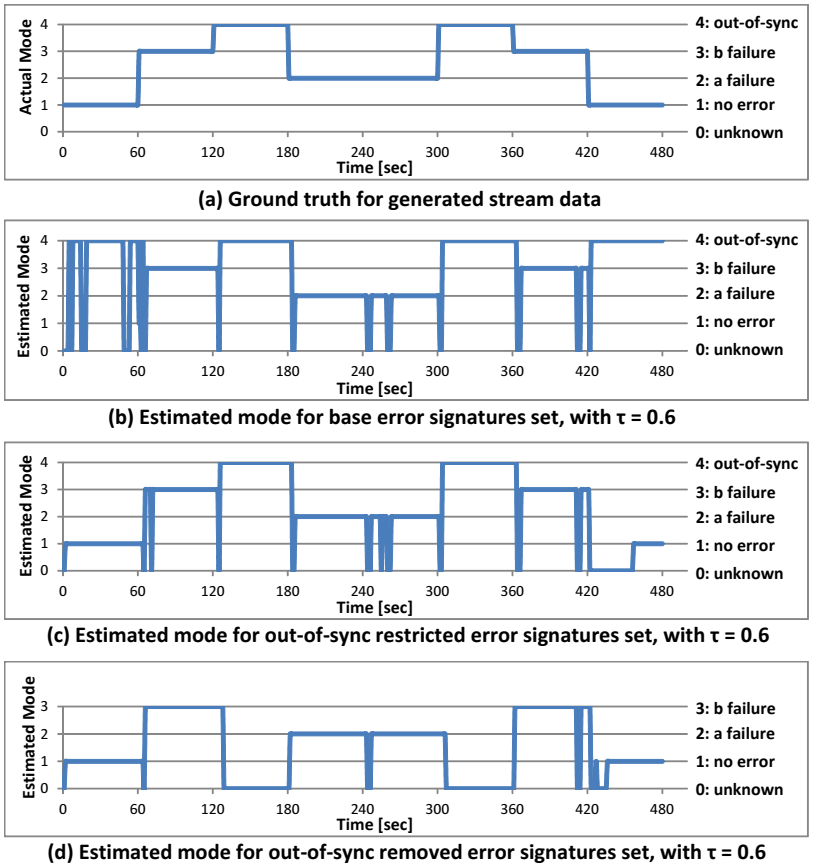

Figure 4: Estimated modes for twice example

4. Performance Metrics and Experimental Results

We evaluate the performance of the proposed error detection algorithm which depends on the window size, ω, representing how much historical data we consider, and the minimum likelihood threshold τ, representing how well data must match a single signature in order to select a corresponding operation mode.

4.1. Performance Metrics

- **Accuracy**: This metric is used to evaluate how accurately the algorithm determines the true mode. Assuming the true mode transition $m(t)$ is known for $t = 0, 1, 2, ..., T$, let $m'(t)$ for $t = 0, 1, 2, ..., T$ be the mode determined by the error detection algorithm. We define $accuracy(m, m') = \frac{1}{T}\sum_{t=0}^{T} p(t)$, where $p(t) = 1$ if $m(t) = m'(t)$ and $p(t) = 0$ otherwise.
- **Average Response Time**: This metric is used to evaluate how quickly the algorithm reacts to mode changes. Let a tuple (t_i, m_i) represent a mode change point, where the mode changes to m_i at time t_i. Let $M = \{(t_1, m_1), (t_2, m_2), ..., (t_{N_1}, m_{N_1})\}$ and $M' = \{(t'_1, m'_1), (t'_2, m'_2), ..., (t'_{N_2}, m'_{N_2})\}$ be the sets of true mode changes and detected mode changes respectively. We compute the average response time as shown in Algorithm 1.

input : True mode changes $M = \{(t_1, m_1), (t_2, m_2), ..., (t_{N_1}, m_{N_1})\}$, t_{N_1+1} = simulation end time,
Detected mode changes $M' = \{(t'_1, m'_1), (t'_2, m'_2), ..., (t'_{N_2}, m'_{N_2})\}$
output: Average response time $AvgResp$

$Responses = 0$;
for $i \leftarrow 1$ **to** N_1 **do**
  Find the smallest t'_j such that $(t_i \leqslant t'_j) \wedge (m_i = m'_j)$; if not found, $t'_j \leftarrow t_{i+1}$;
  $Responses = Responses + (t'_j - t_i)$;
end
return $AvgResp = Responses/N_1$;

Algorithm 1: Average response time computation

4.2. Experimental Results

Based on the metrics defined in the previous section, we evaluate the monitored error for the twice example in Figure 3(a) with five different random seeds for noise generation. We use each of the error signature sets for evaluation: *base*, *out-of-sync restricted*, and *out-of-sync removed.* For the *base* and *out-of-sync restricted* error signature sets, a set of true mode changes is given by $M = \{(1, 1), (61, 3), (121, 4), (181, 2), (301, 4), (361, 3), (421, 1)\}$. However, for the *out-of-sync removed* error signature, we replace the *out-of-sync* errors with *unknown* errors (respectively 4 and 0 on the y-axis of Figure 4) in M. We do this for fairness because the *out-of-sync removed* error signatures cannot detect an *out-of-sync* error at all. To find out the effect on the accuracy and average response time by the window size ω and threshold value τ, we measure these metrics for window size $\omega \in \{5, 10, 15, 20\}$ and threshold $\tau \in \{0.2, 0.4, 0.6, 0.8\}$.

Accuracy and average response time results for the *base*, *out-of-sync restricted*, and *out-of-sync removed* error signature sets are shown in Figure 5. For all the three error signature sets, there is a trend that the accuracy and average response time improve as the threshold τ increases. This result can be explained by the following: there are some cases in which there are two competing modes whose likelihood values are close to each other, and due to the closeness, the mode detection algorithm tends to regard it as an unknown error mode. Higher threshold values are more permissive, thus give better results in this example. However if the choice of τ is too large then this system may choose to enter a known error mode when the correct choice is actually unknown error mode.

There is a positive correlation between the window size and average response time for all the threshold values. This is an intuitive result: the less the algorithm uses past data, the more responsive it becomes to mode changes. Also, a faster average response time leads to a better accuracy result since the error detection algorithm cannot predict mode changes, but only react to them. That is, a smaller window size implies better accuracy. In fact, the *base* and *out-of-sync restricted* error signature sets take the best accuracy/average response time when the window size is the smallest $\omega = 5$ and threshold $\tau = 0.8$. On the other hand, in the case of the *out-of-sync removed* error signature set, the accuracy and average response time are peaked when window size $\omega = 10$. Thus, the most appropriate window size is different depending on each error signature set.

The *out-of-sync removed* error signature set works best. By analyzing three different sets of error signatures for this simple example, we see the importance of the error signature set to get accurate mode estimation results quickly. Especially, as we can see in the results from the *base* error signatures set, error signatures should not be very close in terms of error patterns they match, otherwise those error signatures are vulnerable to noise. *Well-behaved*

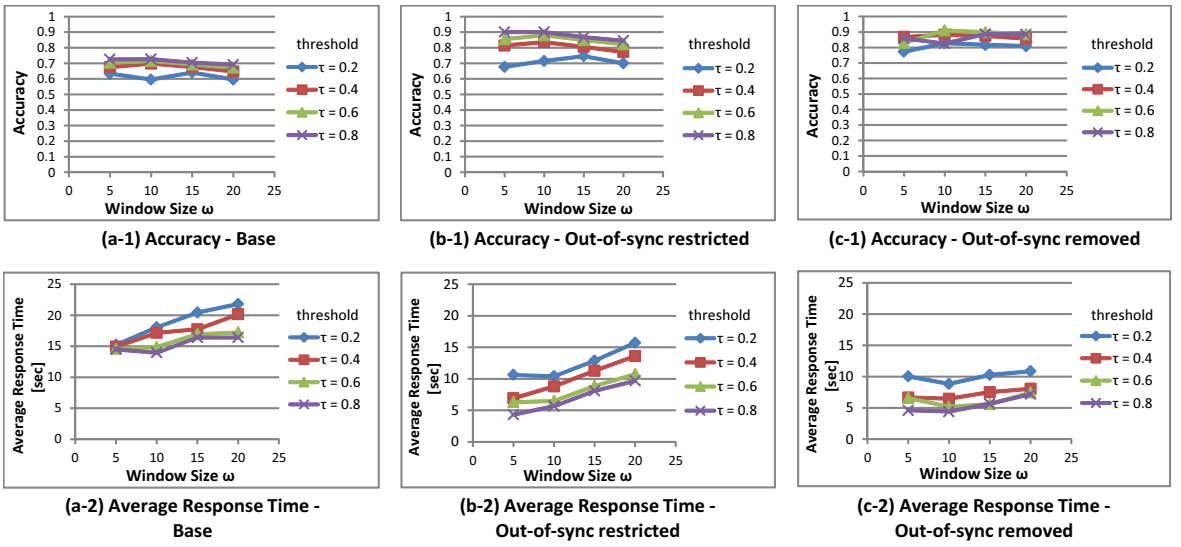

Figure 5: Results of accuracy and average response time for *base*, *out-of-sync restricted*, and *out-of-sync removed* error signature sets

error signatures are those that, under *normal* and *known* error conditions, produce near orthogonal mode likelihood vectors.

5. Implementation with A Spatio-Temporal Data Streaming Programming Language

PILOTS (ProgrammIng Language for spatiO-Temporal data Streaming applications) is a programming language specifically designed for analyzing data streams incorporating space and time, as in applications running on moving objects [3, 2]. Using PILOTS, application developers can easily program an application that handles spatio-temporal data streams by writing a high-level declarative program specification. The system architecture for applications implemented in the PILOTS programming language is shown in Figure 1: everything outside of the dotted box. In this architecture, the application gets data $(d'_1, d'_2, \ldots, d'_n)$ from the *data selection* module. This takes incoming heterogeneous spatio-temporal data streams $(d_1, d_2, \ldots, d_n)$ and outputs homogeneous data streams depending on the current location and time, and the application generates output $(o_1, o_2, \ldots, o_m)$ and data errors $(e_1, e_2, \ldots, e_l)$ based on an *application model*. Whereas spatio-temporal data is available with various spatial density and time frequency depending on data sources, applications often need to process data at a constant frequency. To view such heterogeneous data streams as *homogeneous* data streams, the data selection module specifically provides first-class support for data selection and interpolation so that applications can get data consistently regardless of the data's original spatio-temporal heterogeneity.

We extend the PILOTS programming language to incorporate an error correction method. Two new keywords, `signatures` and `correct`, are introduced in addition to the existing PILOTS grammar defined in [3] to specify which data streams have an associated redundancy and how to correct the incoming data. The statements under the `signature` keyword describe the application's error signature set. Each statement has a label containing any constant parameters, a functional description of the error signature, and an optional list of constraints on the constant parameters separated by commas. The statements under the `correct` keyword declare the relationship between a particular error signature, the corresponding erroneous stream, and the redundancy available to fix the error. This information is enough to know how to handle recoverable error modes. If a data error is detected when matching a known error signature, we can correct an erroneous input as specified under `correct`. If a signature is not included in the `correct` clause, then it is a *known* but *unrecoverable* error. Here we explain how error corrections can be written in the program specification by using the *Twice* example, as shown in Figure 6. The error correction support for PILOTS is realized by the *error detection* module, depicted in Figure 1, which takes all the error outputs $(e_1, e_2, ..., e_l)$ and tries to detect erroneous data inputs by comparing the error outputs with the *known error signatures*. If an error

on data input d_i' is detected, it will be replaced by the value specified in the `correct` clause by the *error recovery* module.

```
program twice;
    inputs
        a: (t) using closest(t);
        b: (t) using closest(t);
    outputs;
    errors
        e: (b - 2 * a) at every 1 sec;
    signatures
        s0:    e = 0;
        s1(k): e = 2*t + k;
        s2(k): e = -2*t + k;
        s3(k): e = k, abs(k) > 20;
    correct
        s1(k): a = b / 2;
        s2(k): b = 2 * a;
end;
```

Figure 6: A simple program specification with error correction

6. Related Work

First and foremost this work builds upon the programming language PILOTS [2, 3] which targets spatio-temporal data streaming applications such as those found in flight systems. The detailed investigation reported in [1] suggests that our notion of error signatures to detect data errors can be quite useful. The concept of the moving object data base (MODB) which supports spatio-temporal data streaming is discussed in [4]. This research is relevant because many applications of error signatures will include data corresponding to a moving object (such as a plane).

Stream processing has become very attractive in the last decade. Surveys on general data streaming applications and methods include [5, 6]. The concept of the rule engine is discussed in [5] which has many similarities to our error signatures system but does not correct the input streams. General-purpose data stream management systems [7, 8, 9] cannot afford the declarative specification of data streams and data error correction that our domain-specific approach provides. In [10] a component is added to stream processing systems to *orchestrate* the behavior of the applications, including correcting domain specific errors. However this is an event driven system, the input data is not being directly monitored. To incorporate error correction using the notion of error signatures into a distributed environment, the work presented in [11] for setting up a set of distributed stream processing systems may be useful. A distributed data processing framework can help with the performance and scalability of data analyses.

7. Conclusion

In this paper we devised a multi-modal data error detection and recovery architecture based on our definition of *error signatures* as mathematical function patterns. We defined *mode likelihood vectors* as a quantifiable measure of the likelihood of the application being in a normal or a particular error mode, as defined by an error signature. Well-behaved error signatures are those that produce orthogonal mode likelihood vectors on *normal* and *known* error conditions. Ill-defined error signatures (those producing non-orthogonal vectors) lead to more undesirable or incorrect *unknown error* mode conditions, rendering our error detection and correction framework less useful. Real-time analysis of error streams and pattern matching against known error signatures enables streaming applications to switch from normal operation mode into *known error modes*. If the known error is *recoverable*, thanks to the redundancy available in the data, we autonomously correct the faulty data stream, so that applications continue to behave normally. Furthermore, we continue monitoring the input streams, so that normal operation can be reinstated when data are considered no longer erroneous.

Accuracy and responsiveness depend on the *window size*, ω, of the monitored data, and on the *threshold*, τ, imposed on the relative likelihood of a mode before accepting a change in the application's mode of operation. Using a

simple streaming application we found that, the larger ω is, the less responsive (higher response time) the algorithm. However if ω is too small, the system enters *unknown* mode more frequently affecting both accuracy and responsiveness. When the signature set is well-behaved, τ has less effect on accuracy, since mode likelihood vectors will be near orthogonal. However, for less well-behaved signature sets, smaller values of τ will cause the system to enter *unknown* error mode more often, while larger values of τ will produce more false positives. Since, the requirements on accuracy and responsiveness are ultimately application-dependent, application developers need to find the right balance of these parameters to tune their applications' error detection and correction behavior. The implementation of the extended PILOTS programming language, due to its declarative nature, will help quickly prototype new applications and develop better error detection and correction methodologies.

Future work includes creating well-behaved error signatures for aeronautical applications in order to correct redundant data such as air speed or fuel levels. We intend to extend this work to incorporate quantitative logical inference based on spatio-temporal knowledge and constraints to promote autonomous data stream management. A method for enforcing logical constraints within streaming applications is presented in [12]. A comprehensive look at the variations of spatio-temporal logic and their computational complexity is presented in [13]: the dichotomy of qualitative and quantitative logic is discussed with respect to space and time. Further research on spatio-temporal logic and constraint logic programming includes [14, 15].

Acknowledgments

This research is partially supported by the Air Force Office of Scientific Research Grant No. FA9550-11-1-0332.

References

[1] B. d'Enquêtes et d'Analyses pour la Sécurité de l'Aviation Civile, Final Report: On the accident on 1st June 2009 to the Airbus A330-203 registered F-GZCP operated by Air France flight AF 447 Rio de Janeiro - Paris.
URL `http://e1.flightcdn.com/live/special/Air_France_447_AFR447_Final_Report-en.pdf`

[2] S. Imai, C. A. Varela, Programming spatio-temporal data streaming applications with high-level specifications, in: 3rd ACM SIGSPATIAL International Workshop on Querying and Mining Uncertain Spatio-Temporal Data (QUeST) 2012, Redondo Beach, California, USA, 2012.

[3] S. Imai, C. A. Varela, A programming model for spatio-temporal data streaming applications, in: Dynamic Data-Driven Application Systems (DDDAS 2012), Omaha, Nebraska, 2012, pp. 1139–1148.

[4] K. An, J. Kim, Moving objects management system supporting location data stream, in: Proceedings of the 4th WSEAS international conference on Computational intelligence, man-machine systems and cybernetics, CIMMACS'05, World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, 2005, pp. 99–104.

[5] M. Stonebraker, U. Çetintemel, S. Zdonik, The 8 requirements of real-time stream processing, SIGMOD Rec. 34 (4) (2005) 42–47.

[6] B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom, Models and issues in data stream systems, in: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, PODS '02, ACM, New York, NY, USA, 2002, pp. 1–16.

[7] D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. hyon Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, S. Zdonik, The design of the borealis stream processing engine, in: In CIDR, 2005, pp. 277–289.

[8] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, K. Ito, R. Motwani, U. Srivastava, J. Widom, Stream: The stanford data stream management system, Springer, 2004.

[9] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, M. Doo, Spade: the system s declarative stream processing engine, in: Proceedings of the 2008 ACM SIGMOD international conference on Management of data, SIGMOD '08, ACM, New York, NY, USA, 2008, pp. 1123–1134.

[10] G. Jacques-Silva, B. Gedik, R. Wagle, K.-L. Wu, V. Kumar, Building user-defined runtime adaptation routines for stream processing applications, Proc. VLDB Endow. 5 (12) (2012) 1826–1837.

[11] M. Branson, F. Douglis, B. Fawcett, Z. Liu, A. Riabov, F. Ye, Clasp: collaborating, autonomous stream processing systems, in: Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware, Middleware '07, Springer-Verlag New York, Inc., New York, NY, USA, 2007, pp. 348–367.

[12] A. Lallouet, Y.-C. Law, J. H.-M. Lee, C. F. K. Siu, Constraint programming on infinite data streams., in: T. Walsh (Ed.), IJCAI, IJCAI/AAAI, 2011, pp. 597–604.

[13] F. Wolter, M. Zakharyaschev, Qualitative spatio-temporal representation and reasoning: A computational perspective, in: Exploring Artifitial Intelligence in the New Millenium, Morgan Kaufmann, 2001, pp. 175–216.

[14] R. Gennari, Temporal reasoning and constraint programming - a survey, CWI Quarterly 11 (1998) 3–163.

[15] A. Raffaeta, T. W. Fralhwirth, Spatio-temporal annotated constraint logic programming., in: PADL'01, 2001, pp. 259–273.