TDATA — Test Data Analytics Toolbox

by

Chun-Kai Hsu and Fan Lin

ckhsu@ece.ucsb.edu

fan_lin@umail.ucsb.edu

Department of Electrical and Computer Engineering

University of California Santa Barbara

Santa Barbara, CA 93106

September, 2016

Contents

1 Introduction

There are two major requirements for a software tool to manage very-large-scale integration (VLSI) test data. VLSI test data have a unique data hierarchy: a product consists of lots; a lot consists of wafers; a wafer consists of dies. Moreover, test data have unique properties of internal connection: a die is tested at multiple test stages; a test stage performs multiple test items; a measurement from a test item is evaluated by specification limits. Therefore, the first requirement of a tool designed for analyzing such test datasets is to provide functions of data parsing, recording, selection, and management dedicated to the unique hierarchy and properties.

The other requirement is the flexibility for the integration of multiple research projects. Different projects may utilize test data from different sources and may conduct different learning algorithms. For example, the test time reduction methodology in [1] exploits two statistical learning algorithms for exploring spatial correlations and inter-test-item correlations, respectively. In addition, the silicon characterization framework proposed in [2] utilizes test data from two datasets, which are in different formats and are from different sources.

There are no opensource or public tools can fulfill these two requirements. Therefore, we build our own toolbox that is dedicated to test data analytics developed in the course of this research. The toolbox has the following features: *a*) a modular and object oriented design using MATLAB and Python, *b*) scalable and configurable parser for reading test data stored in different formats, *c*) data pre-processing functions, such as normalization and outlier removal, *d*) data processing functions, such as sampling, splitting, and concatenation, and *e*) various learning algorithms, such as variation modeling, clustering, and classification.

The toolbox can be found at `http://cadlab.ece.ucsb.edu/tdata`. Section 2 of this manual presents the modular architecture of the toolbox, and Section 3 details the data structure of the toolbox. We provide several examples with the toolbox source code.

2 Implementation

This section presents the main functions provided by the toolbox with implementation details.

2.1 Test Data Parsing

Figure 1 illustrates the process flow for parsing test data from the specified sources. The typical sources of test data are files and each file stores the test data from a single wafer. After receiving a request, the process searches for the corresponding saved `Archive` from the local disk storage first. An `Archive` is an object used to store parsed test data and is described in Section 3.2.3. If the test data of interest have been parsed before, an `Archive` might already be saved on the disk based on the user's configurations.

The process then invokes functions of a configured parser module, which is inherited from the base parser module, to read test data from the source file if the saved `Archive` is not existed. The base parser module includes several basic but configurable functions to parse test data in pre-defined formats, such as standard test data format (STDF) or comma-separated values (CSV). The user can configure the parser or even add any enhanced functions to the inherited parser module for accessing specific data format. For instance, a user can add functions to access test data from an online database instead of local files.

The next step, the process creates a new `Archive` for storing and tracking the requested test data. Based on the configurations provided by a user, this newly created `Archive` may be saved to the disk for speeding up the next requisition. Finally, the process returns the parsed test data through `Archive` objects. Note that an `Archive` is used for the test data that belong to one single wafer at one test stage.

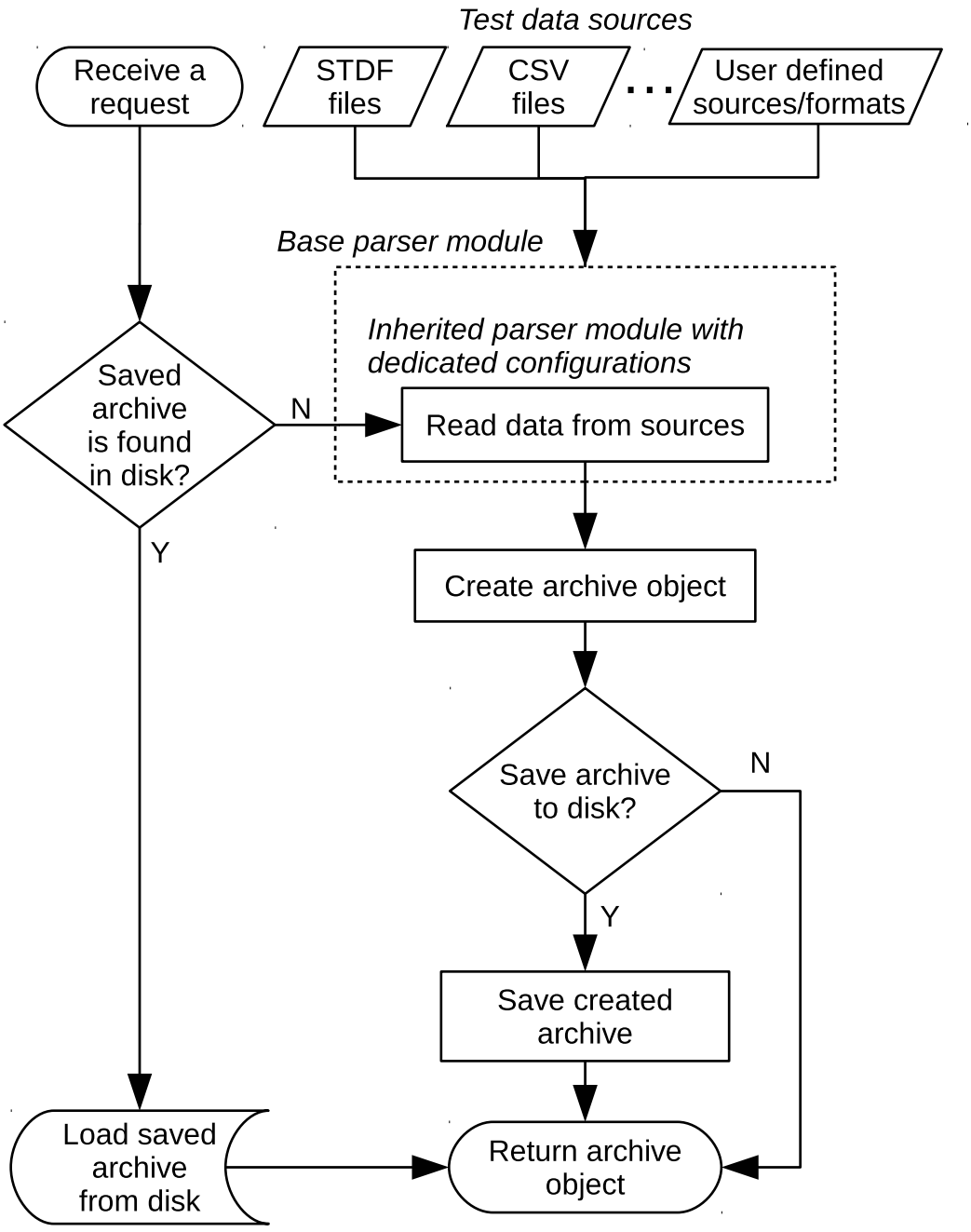

Figure 1: The flow of parsing test data from sources.

2.2 Test Data Selection

In this toolbox, any properties related to a source of test data are recorded by a `Source` object, which is detailed in Section 3.2.8. A user have to access test data through the selection functions of a `Source` that is linked with a particular parser (for the specified test data formats and sources). The toolbox tries to satisfies most of the scenarios in which a user will select test data. The supported scenarios are listed below.

a) Selecting test data based on different hierarchy. A user can retrieve a subset of dies based on lots and/or wafers and/or die locations specified by a user defined list.

b) Selecting test data based on different die types. A user can retrieve a subset of dies based on the specified die types: pass, fail, missing value (dies with missing measurements), invalid (dies with corrupt or damaged measurements), or a user defined list.

c) Selecting test data by sampling. A user can retrieve a subset of dies randomly sampled from a wafer based on a fixed number or a percentage.

d) Selecting test data based on different test item types. A user can retrieve a subset of test measurements based on the specified test item types: parametric, nonparametric/functional, valid (test items with valid spec limits as the principles for pass/fail decisions), or a user defined list.

e) A combination of the above four scenarios. For example, a user can retrieve a subset of test data that consists of 10% randomly sampled fail dies from every wafers in the first and second lots with measurements of every parametric test items.

2.3 Learning Algorithms

The toolbox comes with various build-in learning algorithms, such as the VP and JVP algorithms that are described in [3, 4]; the GL and WGL algorithms that are described in [1]. The toolbox utilizes other projects and tools, such as LIBSVM [5] and MOSEK [6], for efficient data analysis and provides a universal application programming interface (API) for every supported learning algorithm. A user can invokes the same set of methods, such as `train()`, `validate()`, `predict()`, `show()`, and `reset()` to perform different learning algorithms.

3 Architecture and Data Structure

3.1 Configurations

3.1.1 `ConfigBase` Class

`ConfigBase` contains general configurations used in this toolbox.

Attributes:

`fileType` indicates the format/type of test data source files.

`validityIndex` is the index of the column of a test data source file that indicates pass/fail/invalid dies.

`xIndex` is the index of the column of die x coordinates.

`yIndex` is the index of the column of die y coordinates.

`xOffset` indicates the offset of each die x coordinate, so that a x coordinate starts at 1.

`yOffset` indicates the offset of each die y coordinate, so that a y coordinate starts at 1.

`xSize` indicates the maximum value of a die x coordinate (after adding `xOffset`).

`ySize` indicates the maximum value of a die y coordinate (after adding `yOffset`).

`siteIndex` is the index of the column of test site (i.e., test head index in multi-head test setting).

`stepNumIndex` is the index of the column of step number (i.e., the touch down sequence).

`beginIndex` is the index of the column of the first test item.

`columnNum` is the total number of columns in a test data source file.

`titleHeaderNum` indicates the number of lines (headers) before the title of each column in a test data source file.

`dataHeaderNum` indicates the number of lines between the title and the data in a test data source file.

`specFileName` indicates the name of the file with test specifications.

`mapFileName` indicates the name of the file with the mapping of etest (WAT or WET) sites to die coordinates.

`fileDir` indicates the directories of test data source files.

`saveMat` is set to `True` for saving parsed `Archive`s (in mat format) to disk.

`loadMat` is set to `True` for loading saved `Archive`s from disk whenever the mat files are available instead of re-parsing source files.

`matFileDir` indicates the directory for saving parsed `Archive`s.

`maxItemNum` indicates the maximum number of test items that will be used in later. Set `maxItemNum` to 0 for using all test items.

`noValueThreshold` indicates the percentage threshold for excluding test items for which too many dies have no measurements. For example, a test item for which 20 or more out of a total 100 dies have no measurements will be excluded from later process if `noValueThreshold` is set to 20.

`uniqueThreshold` indicates the number threshold for excluding test items for which too many dies have repeat measurements. For example, a test item with less than 5 different measurements will be excluded from later process if `uniqueThreshold` is set to 5.

`excludeTitle` indicates the strings for excluding certain test items. Any test item that have a title with string in `excludeTitle` will be excluded from later process.

`allPass` is set to `True` for neglecting all test item specifications.

`normalizeSite` is set to `True` for adjusting the normalized values of dies if in a multi-site test setting.

3.1.2 `ConfigAlgBase` Class

`ConfigAlgBase` contains configurations used in every learning algorithms.

Attributes:

`preprocessMethod` indicates the method for preprocessing test data: 0 as no preprocess, 1 as feature standardization, and 2 as pattern normalization.

`transformMethod` indicates the method for transforming features: 0 as no transform, 1 as canonical transform, 2 as PCA, and 3 as PCA then canonical transform.

`featureType` indicates the types of features that will be used in learning algorithms. `featureType` can be the combination of the following: 'MM', 'BF', 'MF', and 'VP' which are measurement mean, bilateral filter (spatial pattern), median filter (spatial pattern), and virtual probe (spatial pattern), respectively.

`weightPositive` indicates the weight for positive class in a learning process.

`weightNegative` indicates the weight for negative class in a learning process.

3.2 `data` Namespace

`data` namespace includes data types and objects for parsing, recording, selecting, and managing test data.

3.2.1 `DieFlag` Enumeration

`data.DieFlag` is the enumeration object used for indicating the type of a die or a measurement. A measurement can be described only by `INVALID`, `PASS`, `FAIL`, and `MISSING` enumerators.

Enumerators:

`INVALID` indicates an unreliable or not exist die/measurement.

`PASS` indicates a good die, or a passed measurement.

`FAIL` indicates a bad die, or a failed measurement.

`MISSING` indicates a missing measurement, or a bad die with missing measurements.

`MISSING_PASS` indicates a good die with missing measurements.

`ESCAPE` indicates a bad die that is classified as an test escape.

`ESCAPE_PASS` indicates a good die that is classified as an test escape.

`PASS_USER` indicates a good die based on user defined criteria.

`FAIL_USER` indicates a bad die based on user defined criteria.

`ESCAPE_USER` indicates a test escape (also a bad die) based on user defined criteria.

Member Functions:

`isValid()` returns `false` for a `INVALID` enumerator, and returns `true` otherwise.

`isFail()` returns `true` for `FAIL`, `ESCAPE`, `FAIL_USER`, and `ESCAPE_USER` enumerators, and returns `false` otherwise.

`isPass()` returns `true` for `PASS`, `MISSING_PASS`, `ESCAPE_PASS`, and `PASS_USER` enumerators, and returns `false` otherwise.

`isEscape()` returns `true` for `ESCAPE`, `ESCAPE_PASS`, and `ESCAPE_USER` enumerators, and returns `false` otherwise.

`isUserDefine()` returns `true` for `PASS_USER`, `FAIL_USER`, and `ESCAPE_USER` enumerators, and returns `false` otherwise.

`isMissing()` returns `true` for `MISSING` and `MISSING_PASS` enumerators, and returns `false` otherwise.

3.2.2 `PackageType` Enumeration

`data.PackageType` is the enumeration object used for indicating the type of values stored in `data.Package` objects, which is discussed in Section 3.2.6.

Enumerators:

`ORIGINAL` indicates the raw data directly parsed from a test data source.

`NORMALIZED` indicates the normalized data (from the original test data).

`RESIDUAL` indicates the residual of test data, i.e., the difference between two data with the same dimension.

`TRANSFORMED` indicates that the data have been transformed by a statistical procedure, such as PCA.

3.2.3 `Archive` Class

`data.Archive` is the object for storing and tracking the parsed test data of a single wafer. An `Archive` can be saved on disk in Matlab MAT-file format and be loaded into memory while being requested by other objects or functions. The raw test data are stored unordered in a two-dimensional matrix, `Archive.valueOriginal`, where the first and the second dimensions denote different dies and different test items, respectively. Most attributes of `Archive` are arrays with `dieNum` elements where each element corresponds to a die based on the same order of dies in `valueOriginal`.

Attributes:

`dieNum` indicates the number of stored dies.

`itemNum` indicates the number of test items, i.e., the number of test measurements for one single die.

`valueOriginal` stores the original raw test data.

`valueNormalized` stores the normalized (with respect to a test item) test data.

`validity` indicates the type of each die. The order of elements in `validity` is the same as the order of dies in `valueOriginal` and `valueNormalized`.

`linearIndex` represents the linear index for the location of each die with respect to a wafer map with `waferSize` dimension. It can be transformed into a pair of two-dimensional coordinates using method `Archive.coordinate()`.

`stepNum` represents the number of step (step sequence) for testing a die.

`site` represents the site index for testing a die in a multi-site testing setup.

`waferSize` indicates the size of the wafer for the test data.

`rMean` represents the robust mean of measurements for each test item. The robust mean is calculated based on dies on the same wafer after removing outliers.

`rStd` represents the robust standard deviation for each test item with outlier dies excluded.

`rMedian` represents the robust median for each test item.

Member functions:

`Archive(dieNum, itemNum)` The constructor return an `Archive` object for recording the test data of a wafer with `dieNum` dies and `itemNum` test items.

`[mask, num] = get_validity(flag)` The function returns a mask (a Boolean array) that indicates the dies with `validity` specified by `flag` (a `data.DieFlag`). `num` is the number on ones (trues) in `mask`.

`[mask, num] = get_pass()` The function returns a mask and the number of pass dies.

`[mask, num] = get_fail()` The function returns a mask and the number of fail dies.

`validity = get_itemized_validity(obj, item)` The function returns the validity of each die for the specified test item.

`wafer = get_wafer(itemIndex, dieMask, valueType)` The function returns a two-dimensional wafer map that consists of test measurements of a test item specified by `itemIndex`. If `dieMask` is provided, the wafer map contains specified dies only. The type of values of the wafer map is specified by `valueType`, which is either `data.PackageType.ORIGINAL` or `data.PackageType.NORMALIZED`.

`dieMask = get_dieMask(dieType)` The function returns a mask of dies with type specified by `dieType`, which is a cell array of strings with any combination from `'all'`, `'good'`, `'bad'`, and `'escape'`.

`[x, y] = coordinate(dieIndex)` The function returns the two-dimensional coordinates for a die specified by `dieIndex`.

3.2.4 `Item` Class

`data.Item` records properties related to a test item.

Attributes:

`sourceIndex` represents the index of a test item in the original source file.

`title` represents the title of a test item.

`lowerLimit` represents the lower specification limit of a test item.

`upperLimit` represents the upper specification limit of a test item.

`order` represents the test order of a test item.

`isValid` indicates a test item whether has valid specification limits when evaluating pass/fail of dies.

`isUse` indicates whether the measurement of a test item will be used in other procedures and/or analysis.

`isParametric` indicates whether a test item is parametric, i.e., a test item has parametric test measurements.

Member functions:

`get_limit()` returns both the upper spec limit and the lower spec limit of a test item.

3.2.5 `Sample` Class

`data.Sample` records the indexes of dies sampled from a particular wafer, and the algorithm and configurations used for sampling. `data.Sample` also has attributes for controlling the sampling process. For example, `sampleByStep` indicates that dies are sampled based on probing step number (only applicable while using multi-head tester), and `sampleGoodOnly` indicates that only good (pass) dies are sampled.

3.2.6 `Package` Class

`data.Package` is the object for storing a set of selected dies with measurements of selected test items. The possible scenarios for selecting test data are discussed in Section 2.2. `data.Package` also records the indexes of `data.Archive`'s of the selected dies.

Member functions:

`append(sample, archive)` appends the dies in `archive` based on the recoded indexes in `sample`.

`replace(start, value)` replaces measurements with `value` starting from the `start`-th die.

`remove(list)` removes a set of stored dies based on the indexes specified by `dList`.

`verticalCat(package)` vertically concatenates data in `package` to itself.

`horizontalCat(package)` horizontally concatenates data in `package` to itself.

`subset(dList, iList)` returns a new `data.Package` based on a set of dies specified by `dList` with the measurements of the test items specified by `iList`.

3.2.7 `Parser` Class

`data.Parser` read test data in a particular format defined by the provider of the test data. The details of parsing test data form the specified sources are discussed in Section 2.1. `data.Parser` has various attributes that can be configured for different formates of test data.

Member functions:

`load_data(file)` loads test data from the source specified by `file` and returns a `data.Archive`.

`load_item()` loads attributes of each test item and returns a set of `data.Item`'s where a `data.Item` records the attributes of a test item.

3.2.8 `Source` Class

`data.Source` is linked with a particular parser for the specified test data formats and sources. `data.Source` hides the details of accessing test data in different formats from the user through providing a universal API. The main object of `data.Source` is to support various scenarios for selecting test data based on a combination of multiple user specified configurations.

Attributes:

`saveMat` is set to `True` for saving parsed `Archive`s (in mat format) to disk.

`loadMat` is set to `True` for loading saved `Archive`s from disk whenever the mat files are available instead of re-parsing source files.

`matFileDir` indicates the directory for saving parsed `Archive`s.

`maxItemNum` indicates the maximum number of test items that will be used in later. Set `maxItemNum` to 0 for using all test items.

`noValueThreshold` indicates the percentage threshold for excluding test items for which too many dies have no measurements. For example, a test item for which 20 or more out of a total 100 dies have no measurements will be excluded from later process if `noValueThreshold` is set to 20.

`uniqueThreshold` indicates the number threshold for excluding test items for which too many dies have repeat measurements. For example, a test item with less than 5 different measurements will be excluded from later process if `uniqueThreshold` is set to 5.

`excludeTitle` indicates the strings for excluding certain test items. Any test item that have a title with string in `excludeTitle` will be excluded from later process.

`allPass` is set to `True` for neglecting all test item specifications.

`normalizeSite` is set to `True` for adjusting the normalized values of dies if in a multi-site test setting.

Member functions:

`Source(config, parser)` The constructor return an `Source` object based on the configurations in `config` and the source file `parser`.

`[archive] = load_data(filePtr, dirIndex)` The function loads test data from a file by `filePtr`, which can be either a string or an integer. The integer indicates the index for a file in `Source.files`. If `filePtr` is a string as a file name, user should give `dirIndex` to indicate the directory in `Source.fileDir` where the targeted file is.

`update(filePtr, dirIndex)` The function sets attributes for each test item based on a user specified test data source file for later use. Run `update()` once before any call of `load_data()`.

`examine_die(archive)` The function sets the type of each die stored in an `archive` based on user defined configurations and spec limits. The results may be different from the types stored in the original source files.

`[dieSet] = retrieve(sampleNum, fileIndex, itemType, dieType, valueType)` The function returns a set of test data as a `data.Package` based on the arguments. The details are discussed in Section 2.2.

`unuse_item(itemIndex)` The function sets the inUse flag of a test item to `False`.

`invalid_item(itemIndex)` The function sets the isValid flag of a test item to `False`.

`[sample] = generate_sample(arIndex, sample, num, dieType)` The function returns a set of sampled dies as a `data.Sample` from wafers (`data.Archive`'s) specified by `arIndex` based on the specified `dieType` and number of desired samples. If `num` is equal to 0, sample every die from each targeted `archive`.

`[wafer] = get_wafer(fileIndex, itemIndex, dieType, valueType)` returns a two-dimensional wafer map that consists of test measurements of a test item specified by `itemIndex`.

`[index] = find_test_item(itemName)` The function finds index of test items with `itemName` in the title.

3.3 vp Namespace

vp namespace includes objects and functions of the VP and JVP algorithms described in [3,4].

3.3.1 Config Class

vp.Config provides several configurations. maxTolerance and maxIterNum control the termination conditions of the M-FOCUSS solver used by VP and JVP. sampleNum and clusterNum define the number of sampled dies and the number of jointed test items, respectively.

3.3.2 Model Class

vp.Model records the prediction results of VP/JVP, such as predictable and predictError, which represent the predictability and the prediction error of a parametric test item, respectively.

3.3.3 Learner Class

vp.Learner is the main object for providing methods of the VP/JVP algorithm.

Member functions:

train(inPackage) explores the predictability of each parametric test item based on the given test data, inPackage.

solve_vp(sample) constructs a complete wafer map based on the sampled values specified by sample.

`predict(inPackage)` predicts errors and test escapes for the given test data, `inPackage`, based on the training results.

3.4 `svm` Namespace

`svm` namespace includes objects and functions that are used for the feature engineering with canonical analysis described in [7].

3.5 `gl` Namespace

`gl` namespace includes objects and functions of GL and WGL algorithms described in [1].

3.5.1 `Config` Class

`gl.Config` provides several configurations. `useWeight` controls the weight assignment of WGL function. `lambda` defines the λ value.

3.5.2 `Model` Class

`gl.Model` records the prediction results. `predictable` and `predictError` represent the predictability and the prediction error of a parametric test item, respectively. `alpha` records the explored linear correlations among test items.

3.5.3 `Learner` Class

`gl.Learner` is the main object for providing methods of GL/WGL algorithm.

Member functions:

`train(inPackage)` explores the predictability of each parametric test item based on the given test data, `inPackage`.

`solve_gl(inPackage, lambda)` solves the group lasso regression problem based on the given test data and λ, which are specified by `inPackage` and `lambda`, respectively.

`predict(inPackage)` predicts errors and test escapes for the given test data, `inPackage`, based on the training results.

3.6 `fabia` Namespace

`fabia` namespace includes objects and functions that are used for bicluster pattern classification described in [2, 8]. The Matlab implementation are based on the MTBA toolbox [9].

4 Examples

Please refer to `.m` files in the `example` directory of the TDATA source code.

References

[1] C.-K. Hsu, F. Lin, K.-T. Cheng, W. Zhang, X. Li, J. M. Carulli Jr., and K. M. Butler, "Test data analytics — exploring spatial and test-item correlations in production test data," in *Proc. IEEE Int'l Test Conf. (ITC)*, Sep. 2013.

[2] C.-K. Hsu, P. Sarson, G. Schatzberger, F. Leisenberger, J. Carulli, S. Siddhartha, and K.-T. Cheng, "Variation and failure characterization through pattern classification of test data from multiple test stages," in *Proc. IEEE Int'l Test Conf. (ITC)*, Nov. 2016.

[3] X. Li, R. R. Rutenbar, and R. D. Blanton, "Virtual probe: A statistically optimal framework for minimum-cost silicon characterization of nanoscale integrated circuits," in *Proc. IEEE/ACM Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov. 2009.

[4] S. Zhang, F. Lin, C.-K. Hsu, K.-T. Cheng, and H. Wang, "Joint virtual probe: Joint exploration of multiple test items' spatial patterns for efficient silicon characterization and test prediction," in *Proc. Design, Automation, and Test in Europe (DATE)*, Mar. 2014.

[5] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Trans. on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011, software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[6] M. ApS, *The MOSEK optimization toolbox for MATLAB manual. Version 7.1 (Revision 28).*, 2015. [Online]. Available: http://docs.mosek.com/7.1/toolbox/index.html

[7] F. Lin, C.-K. Hsu, and K.-T. Cheng, "Feature engineering with canonical analysis for effective statistical tests screening test escapes," in *Proc. IEEE Int'l Test Conf. (ITC)*, Oct. 2014.

[8] S. Hochreiter, U. Bodenhofer, M. Heusel, A. Mayr, A. Mitterecker, A. Kasim, T. Khamiakova, S. Van Sanden, D. Lin, W. Talloen *et al.*, "FABIA: factor analysis for bicluster acquisition," *Bioinformatics*, vol. 26, no. 12, pp. 1520–1527, Apr. 2010.

[9] J. K. Gupta, S. Singh, and N. K. Verma, "MTBA: Matlab toolbox for biclustering analysis," in *IEEE Workshop on Computational Intelligence: Theories, Applications and Future Directions*, Jul. 2013, pp. 94–97.