Specification-based CSV Support in VDM

The 21st Overture Workshop

10th March 2023

Leo Freitas[1]

Aaron John Buhagiar[2]

1. School of Computing, Newcastle University

2. Translational and Clinical Research Institute, Newcastle University

Introduction

- Standard (RFC4180) data exchange format
- Ubiquitous use in a myriad of domains
  - Data science applications
  - Medical data and embedded
  - Payment systems
- Many variations and versions are used
- VDM standard CSV has limitations
- We created an improved CSV library with:
  - Data validation formal specification
  - Ease-of-use
  - High performance
- Distributed with the VDM Toolkit [1]

[1] https://github.com/leouk/VDM_Toolkit

VDM standard CSV Library

- Slow and error-prone (e.g. low-level IO)
- Line-by-line parsing in VDM
- Limited support for various CSV formats
- Only hard-coded IO native calls available
  - Limits CSV performance and formats variety
- Imported data is of a wildcard (?) type
  - Users have to further introspect meaning
- Different from CSVReader
  - CSV as user data not structural information

Design Principles

Simple

Accurate

Fast

Effective

Types and Parsers

- CSV Lib offers basic types for column typing

```
CSVType = <Integer> | <Float> | <String> | <Boolean>;

CSVValue = int | real | String | bool;
```

- Multiple parses are available

```
CSVParser = <Native> | <Univocity> | <Apache> | <OpenCSV> | <QuirkCSV>;
```

Simple: Ease of Use

- Accessible entry points that abstract from IO native calls
  1) Out-of-the-box setup
  2) Configurable (e.g. CSV settings)
- Allows direct native calls for better extensibility and control (3)
- Data validation can be formally specified

1

```
--@doc load file path with given headers and no invariants: insists on knowking header types.
Launch | Debug
loadSimpleHeadersCSV(path: Path, parserChoice: CSVParser, headerTypes: seq1 of CSVType, s: bool) ==
    (
        let headers: Headers0 =
            [ mk_Header0(
--@doc top-level call: given a file and typed headers (possibly with invariants)
```

2

```
--      loads the CSV as part of the state
Launch | Debug
loadCSV(path: Path, parserChoice: CSVParser, settings: CSVSettings, headersToInstall: Headers0, s: bool) ==
    (
        printf("Loading CSV file %1s\n", [path]);
        if file_status(path) <> <Valid> then
```

3

```
Launch | Debug
csv_read_data: Path * CSVParser * CSVSettings * Headers0 -> bool * Errors * Data0
csv_read_data(path, parser, settings, headers) == is not yet specified
pre file_status(path) = <Valid>
post
    (let mk_(ok, -, mk_Data0(settings', headers', matrix', finv')) = RESULT in
```

Simple: Configurable Setup

- CSV entries strong typing defined through semantic headers as:
  - Column Name
  - Datatype
  - Default Value
  - Cell invariant
  - Column Invariant
- CSV settings define expected file properties as:
  - Presence of blank lines
  - Existence of a header row
  - Comment string

```
EXAMPLE_DATA_HEADERS: Headers =
    [mk_Header0("Name"       , <String> ,         "Joe", nil, COL_INVARIANT_UNIQUE_NAME),
     mk_Header0("Age"        , <Integer>,       MIN_AGE, CELL_INVARIANT_AGE, nil),
     mk_Header0("Weight(Kg)", <Float>  , MIN_WEIGHT_KG, CELL_INVARIANT_WEIGHT, nil),
     mk_Header0("Height(cm)", <Float>  , MIN_HEIGHT_CM, CELL_INVARIANT_HEIGHT, nil),
     mk_Header0("BMI"        , <Float>  ,       MIN_BMI, CELL_INVARIANT_BMI, nil)
    ];
```

Simple: Reporting

- Simple and descriptive reporting
  - Short rows (i.e. not enough columns)
  - Declared type violations (e.g. string for nat)
  - User defined invariant violations
- Provides cell locations for correction
- Striving to have a strongly-typed CSV

```
 successful for "CSVShortRowExample.csv.out"
SV file "CSVInvFailedExample.csv"
p with <Univocity> parser and   5 headers for "CSVInv
 error: ignoring 1 short rows from "CSVInvFailedExampl
(1, 5): "CSV row 1 is too short for header: expected 5
riants failed for "CSVInvFailedExample.csv": 13CSV inv
(1, 1): "Invalid col  invariant: repeated names"
(2, 1): "Invalid col  invariant: repeated names"
(2, 2): "Invalid cell invariant: below minimal age"
(3, 1): "Invalid col  invariant: repeated names"
(3, 2): "Invalid cell invariant: above maximal age"
(3, 3): "Invalid cell invariant: above maximal weight"
(3, 5): "Invalid cell invariant: above maximal BMI"
(4, 1): "Invalid col  invariant: repeated names"
(4, 4): "Invalid cell invariant: above maximal height"
(4, 5): "Invalid cell invariant: below minimal BMI"
(5, 1): "Invalid col  invariant: repeated names"
(6, 1): "Invalid col  invariant: repeated names"
(7, 1): "Invalid col  invariant: repeated names"
 print successful for "CSVInvFailedExample.csv.out"
/ invariant failure at 13 cells:
(1, 1): "Invalid col  invariant: repeated names"
(2, 1): "Invalid col  invariant: repeated names"
(2, 2): "Invalid cell invariant: below minimal age"
(3, 1): "Invalid col  invariant: repeated names"
(3, 2): "Invalid cell invariant: above maximal age"
(3, 3): "Invalid cell invariant: above maximal weight"
(3, 5): "Invalid cell invariant: above maximal BMI"
(4, 1): "Invalid col  invariant: repeated names"
```

Accurate

- Checks on imported data
  - Short rows
  - Declared header type validation
- User defined invariants
  1. Cell invariants
  2. Column invariants
  3. Row invariants
  4. File Invariants
- Lambda abstractions capture invariants as record fields
- Invariants return a **Reason**

```
1  CSVCellInv = (CSVType * CSVValue -> Reason);
```

```
2  CSVColInv = (Header0 * TransposedRow -> Reason);
```

```
3  CSVRowInv = (Headers0 * Row -> Reason);
```

```
4  CSVFileInv = (Headers0 * Matrix -> Reason);
```

Accurate: Cell Invariant

- Invariants that act upon cells directly
- Allows for cell validation
- Examples
  - Upper/lower bounds
  - Cell text validation
  - Specific value enforcement
  - etc…

```
CELL_INVARIANT_AGE: CSVCellInv =
    (lambda -: CSVType, v: CSVValue &
        if v < MIN_AGE then
            "below minimal age"
        else if v > MAX_AGE then
            "above maximal age"
        else
            nil);

CELL_INVARIANT_WEIGHT: CSVCellInv =
    (lambda -: CSVType, v: CSVValue &
        if v < MIN_WEIGHT_KG then
            "below minimal weight"
        else if v > MAX_WEIGHT_KG then
            "above maximal weight"
        else
            nil);
```

Accurate: Column Invariant

- Column-wide invariant
- Allows for validation per header across rows
- Example
  - Uniqueness
  - Dependence

```
COL_INVARIANT_ORDERED: CSVColInv =
    (lambda h: Header0, c: TransposedRow &
        if not h.type in set  {<Integer>, <Float>} then
            "not number typed"
        else if --not (forall i, j in set inds c & i < j => c(i) <= c(j))
                       (exists i, j in set inds c & i < j and c(i) > c(j)) then
            "column is not in ascending order"
        else
            nil
    );
```

Accurate: Row Invariant

- Row-wide invariant
- Allows for validation per row across all headers

- Examples
  - Consistency
  - Dependence
  - Redundance

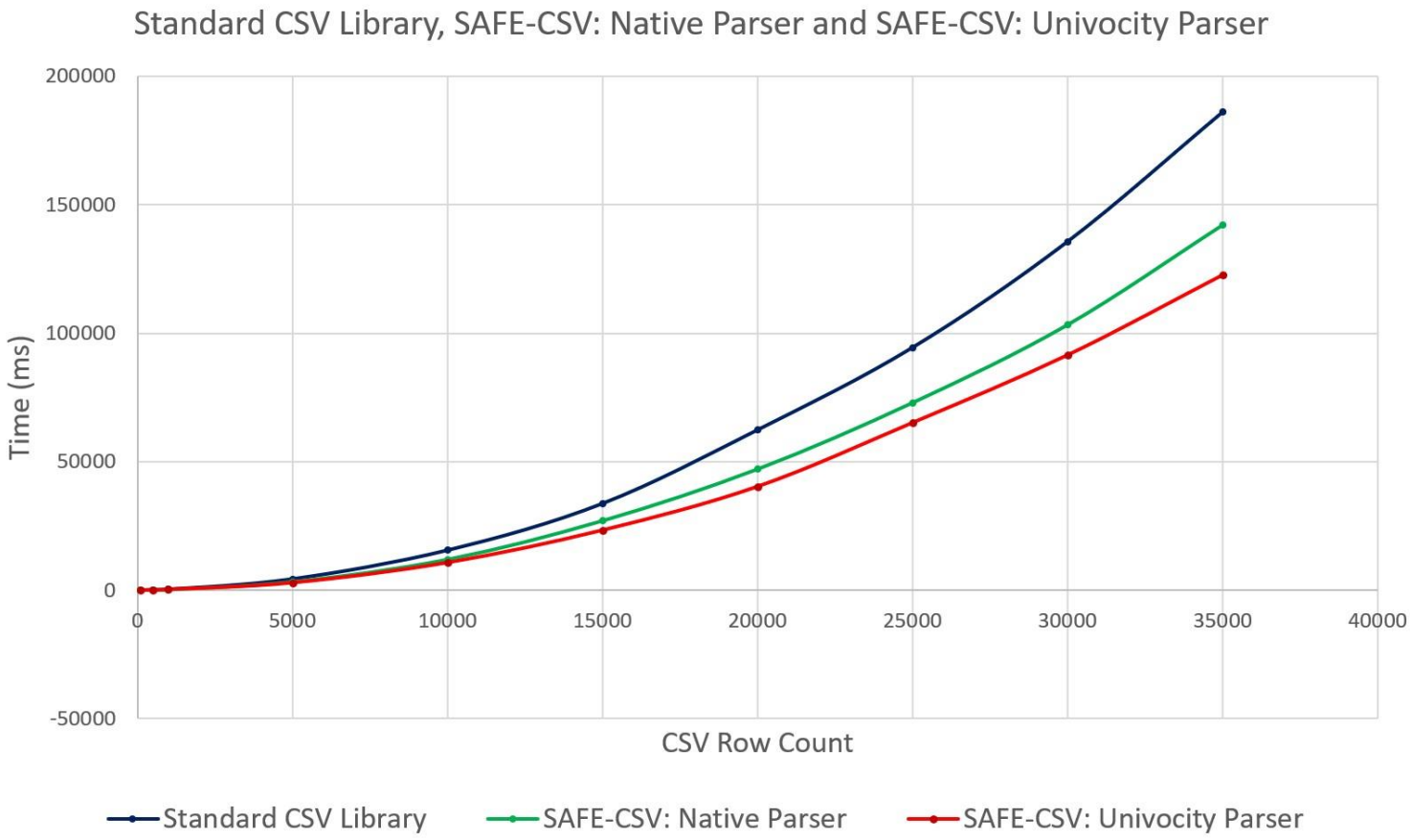
```
ROW_INVARIANT_BMI_CONSISTENCY: CSVRowInv =
    (lambda h: Headers0, r: Row &
        if BMI_ROW_INDEX > len h then
            "invalid BMI header"
        else
            (let bmi: real = calculated_bmi(r) in
                if floor(bmi) = floor(r(BMI_ROW_INDEX)) then
                nil
                else
                "invalid BMI for given CSV weight and height: expected " ^
                    val2seq_of_char[real](bmi) ^"; found " ^ val2seq_of_char[real](r(BMI_ROW_INDEX)))
    );
```

Accurate: File Invariant

- Invariant across all CSV cells
- Example
  - Dependence
  - Redundancy

```
FILE_INVARIANT_CONSISTENCY: CSVFileInv =
    (lambda h: Headers0, m: Matrix &
        if NAME_ROW_INDEX > len h then
            "invalid name header"
        else if AGE_ROW_INDEX > len h then
            "invalid age header"
        else
        (
            let
                dups : set of nat = { j | i, j in set inds m.cells & i <> j
                                                                    and
                                                                    check_name_age_uniqueness(m.cells(i),
                                                                                              m.cells(j))}
            in
                if (dups <> {}) then
                    "duplicate persons found in row(s) " ^ val2seq_of_char[set of nat](dups)
                else
                    nil
```

Fast


Standard CSV Library, SAFE-CSV: Native Parser and SAFE-CSV: Univocity Parser
200000
150000
100000
50000
0
-50000
Time (ms)
0
5000
10000
15000
20000
25000
30000
35000
40000
CSV Row Count
Standard CSV Library
SAFE-CSV: Native Parser
SAFE-CSV: Univocity Parser

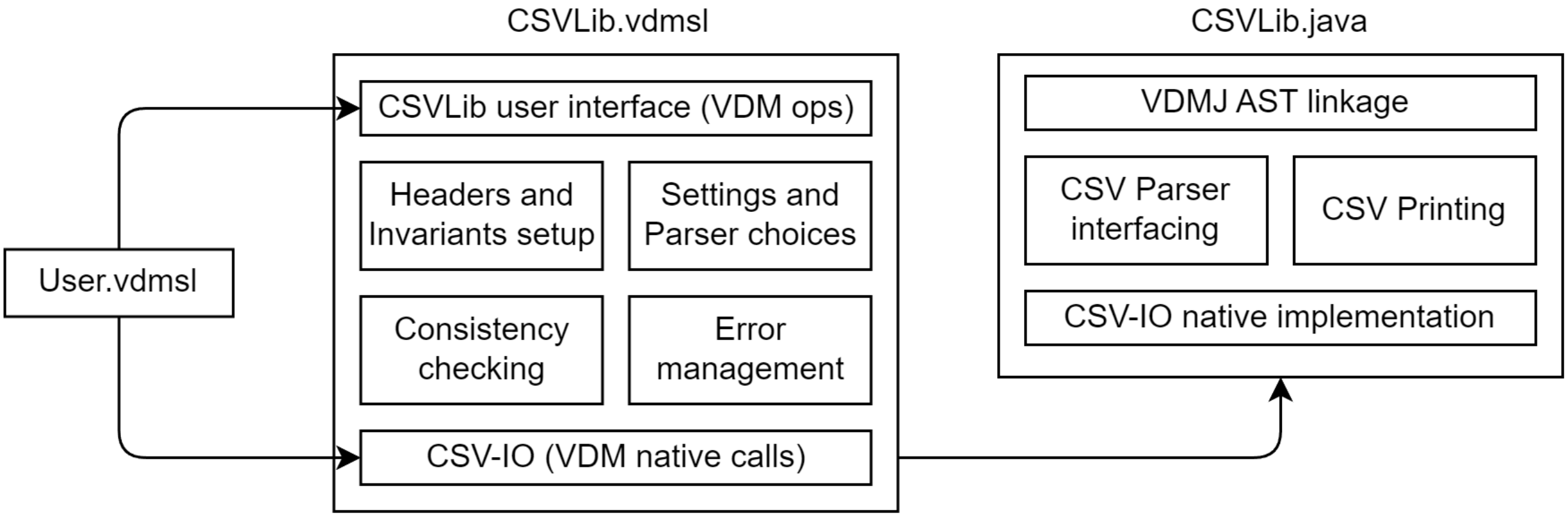
Effective

- Multiple CSV format variants
  - Better tolerance to CSV format variability (e.g. CSV UTF8, MS-DOS, Mac, etc.)
  - Delegate CSV format type to parsers (e.g. formalisation of CSV format itself)
- Ease of use: use of VDM state and operations as entry points
- Improved validation, error handling and reporting
- Faster performance through multiple CSV parsers

Effective

- Tested on multiple CSV format variants
  - CSV UTF8, MS-DOS and Mac
- Applied to multiple domains
  - ScubaTx organ preservation device medical device logs
  - UNOS (United Network for Organ Sharing) lung transplant history logs
  - EMV payment system transaction logs
  - Personalised medicine DSL “certificate of treatment”
  - Neonatal dialyser finite state machine control system definitions
  - Etc.

Library Architecture



Future Work

- Implementation of debugging environment
- Improved variety of CSV formats
  - Nested CSVs
  - CSV Settings
  - More CSV data types
  - Multiple CSV headers per file

Thanks for Listening

Newcastle University