Specification-based CSV support in VDM

Leo Freitas[1]

School of Computing, Newcastle University,
leo.freitas@newcastle.ac.uk

Abstract. CSV is a widely used format for data representing system's control, information exchange and processing, logging, *etc*. Nevertheless, the format is riddles with tricky corner cases and inconsistencies, which lead crucial input data unreliable. Thus, rendering modelling or simulation experiments unusable or unsafe. We address this problem by providing a **SAFE**-CSV VDM-library that is: **S**imple, **A**ccurate, **Fast**, and **E**ffective. It extends an ecosystem of other VDM mathematical toolkit extensions, including a translation and proof environment for VDM in Isabelle [1].

Keywords: VSCode, VDM, CSV, file formats, Libraries

1 Introduction

The Comma Separated Values (CSV) format is widely used for a variety of applications: from data science as inorgan transplant allocation [**?**], embedded systems representation as in state machines for medical devices [**?**, **?**], pharmaceutical applications [**?**], log files on various kinds of systems, databased, payments, government system and so on. Despite being an RFC 4180 standard [2], there are many versions and variations [3]. Its long history of use [4] has inspired various other "simple" formats for data management and exchange, such as JSON [**?**], XML and various spreadsheet formats.

The Vienna Development Method (VDM) has has been widely used both in industrial contexts and academic ones covering several domains of the field: Security [**?**, **?**], Fault-Tolerance [**?**], Medical Devices [**?**], among others. We extend VDM specification support with a suite of tools and mathematical libraries [5]. The work is also integrated within the VDM Visual Studio Code (VS Code) IDE.

In this paper we report on the recent extension of the VDM toolkit to support CSV format parsing, validation and printing. We illustrate its use with a variety of scenarios frequently seen in practical uses of CSV.

[1] https://github.com/leouk/VDM_Toolkit/
[2] TODO
[3] https://commons.apache.org/proper/commons-csv/
[4] https://bytescout.com/blog/csv-format-history-advantages.html
[5] https://github.com/leouk/VDM_Toolkit/

2 Background

CSV is plagued with a variety of fiendish scenarios leading to unsuspecting errors [**?**, **?**, **?**, **?**]. For critical applications relying on the format, this is particularly problematic. This motivated the creation of a formal environment to capture CSV problems clearly and concisely, and to validate outcomes according to user-defined invariants of different nature. For example, data type invariants on CSV cells and consistency invariants on row or column data, such column ordering or row data redundancy consistency (*e.g.* weight, height and BMI info must be correlated). Furthermore, CSV files can often contain implicit defaults or inconsistent correspondences [**?**]. Thus, we are interested in capturing such constraints formally using a VDM library (`CSVLib`).

We integrate this library within the VDM VSCode extension (https://marketplace.visualstudio.com/items?itemName=overturetool.vdm-vscode), as well as VDMJ [**?**].

Related work on EMV2, FMU, CSVReader: TODO

3 Design Principles

Our VDM CSV library has four core design principles:

1. **S**imple: ease of use is crucial, given CSV processing is pervasive and rather menial task;
2. **A**ccurate: CSV input errors accounts for a considerable amount of modelling process inaccuracies; hence, we wanted a solution where multiple forms of expected validation were possible;
3. **F**ast: CSV input can often be large; hence, varied and computationally efficient IO parsing solutions are important [6];
4. **E**ffective: there are multiple CSV format-variants [7]; hence, practical use for a variety of such variations is important.

These **SAFE** principles underpin the overall library design goals. Its architecture is divided in three parts: i) native calls to IO; ii) three distinct CSV invariant checking mechanism per cell, across row and across column; and a series of supporting functions and operations to enable easy access to the CSV functionality.

Moreover, we accept that real CSV applications are riddled with errors and inaccuracies. Thus, we also provide rich error reporting and support for CSV that betrays expected invariants to be processed and accurate information be given to users. Once validation has taken place, users can confidently access the CSV data to their target end, and also print it out (after processing or otherwise) to a file.

[6] https://github.com/uniVocity/csv-parsers-comparison

[7] https://commons.apache.org/proper/commons-csv/

4 Library architecture

Next, we present how the library architecture implements our design principles (see **??**). User models have to import `CSVLib.vdmsl`, which provides CSV-IO (*e.g.,* parsing and printing) as VDM native function calls. VDM native functions are a mechanism for linking VDM specification with an underlying Java implementation servicing each of the VDM native calls defined. For example, VDMJ provides native calls for some trigonometric functions, random value generation, VDM values converstions to string, and so on.

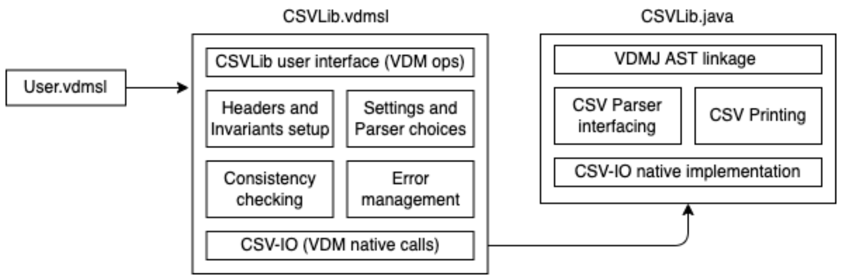
CSVLib.vdmsl
CSVLib.java
User.vdmsl
CSVLib user interface (VDM ops)
Headers and Invariants setup
Settings and Parser choices
Consistency checking
Error management
CSV-IO (VDM native calls)
VDMJ AST linkage
CSV Parser interfacing
CSV Printing
CSV-IO native implementation

Fig. 1: `CSVLib.vdmsl` architecture.

`CSVLib.vdmsl` provides various functionalities including:

1. Settings: line comment, skipping blank lines, multi-entries CSV (with multiple header instances), *etc.*;
2. Parsers: implementations geared for speed, multiple CSV formats, *etc.*;
3. CSV headers: strongly typed and with default values;
4. CSV invariants: checks over cells, across rows and across columns; and
5. Error handling: cell located with explanatory reason(s) for failure.

`CSVLib` use

`CSVLib` has two entry points: either direct calls to CSV-IO native calls, or via operations on a state-based specification of expected library usage. The former provides the most flexible access to CSV functionality, whereas the later provides the most convenient access to CSV functionality. For instance, the simplest use to the CSV library is a call to the operation:

```
loadTrivialCSV(path, <Native>, [<Str>,<Int>,<Bool>], false);
```

This will load the CSV file on `path` using the given parser interface implementation. We currently support two interfaces: our own native implementation and one of the fastest we could find (*e.g.* see https://github.com/uniVocity/csv-parsers-comparison). Other implementations are possible/planned.

The list of quote values contain the types of the expected CSV columns, and the final argument states that processing will not be strict. That means, any failed invariant checks will be tolerated and recorded for further user inspection in the state. Otherwise, if strict was `true`, then invariant failures will not be tolerated and the result is an empty CSV.

CSV headers

Many CSV files contain no header. Nevertheless, they ought to have a notion of type correctness for each value read (*i.e.* an implicit type). In `CSVLib`, headers must have a type and a default value. Defaults are important for common situations like comma-sequences. Types are important to enable type consistency and invariant checking, whereas default values are important so that we can infer what was the hidden intention behind comma-sequences, and to ensure defaults satisfy type invariants. For instance:

```
mk_Header("Age", <Integer>, 18, cell_inv, col_inv)
```

`CSVLib` headers may also contain a description and two invariants: one for every cell underneath a header, and another for the overall column.

CSV invariants

We have three CSV invariants, which are captured by the following VDM function types:

```
CSVCellInv = (CSVType * CSVValue -> Reason);
CSVRowInv  = (Headers * Row -> Reason);
CSVColInv  = (Header  * TransposedRow -> Reason);
```

Invariant checks return a `Reason`, which is either **`nil`** in case the invariant is satisfied, or a non-empty string explaining the cause of failure. This enables the specifier fine-grained control over errors occuring within a CSV file. Cell invariants receive the declared header type and the read cell value to be checked. Row invariants receive all readers and current row of cells to be checked. Column invariants receive the column header and the current col of cells to be checked, viewed as a transposed row.

Some invariant kinds have an implicit check, regardless whether user-defined invariant is given or not. For cells, this check values against their declared header type, whereas for rows, we implicit check that header size corresponds to row size (*e.g.* no short rows); there are no implicit column invariant checks.

For the `Age` header above, users could define `cell_inv` (L1–3) to enforce age limits, whereas `col_inv` (L5) could enforce no age uniqueness as:

```
1    (lambda -: CSVType, v: CSVValue &
2        if v < 18 then "below minimal age" else
3        if v > 65 then "above maximal age" else nil);
4
5    (lambda h: Header, c: TransposedRow & card elems c = len c);
```

Listing 1.1: Cell and Column Invariant Definitions

Row invariants are useful for checking dependencies/redundancies across the row cells. If the CSV file had three extra `<Float>` headers for weight, height and BMI (Body Mass Index), there are interesting checks possible. For instance, given age type range (`{18...65}`) it is possible to presume some minimum height expected (*i.e.* row dependency invariant); and given BMI's formulae ($\frac{height}{weight^2}$), its cell value can be calculated (*i.e.* row redundancy invariant).

CSV VDM native calls

`CSVLib` has four native calls: three functions implementing file status, CSV read and write, respectively; and one operation implementing low-level IO (not CSV) errors.

```
file_status    : Path -> FileStatus;
csv_read_data  : Path * CSVParser * CSVSettings * Headers ->
                 bool * Errors * Data;
csv_write_data: Path * Data -> bool;
lastError      : () ==> Reason;
```

CSV read data implements the link with Java CSV parsers. It expects a valid path, a known parser, settings and valid headers. It returns a tripple. In the result, if the boolean flag is **`false`**, some unexpected Java error occurred that can be inspected by a call to `lastError()`, errors and data are empty. Otherwise, if errors are not empty, then CSV parsing has identified short rows (*e.g.* row's size smaller than header's), and they contain a set of errors where such errors occured in the CSV file. If the boolean flag is **`true`**, then the resulting data will have the CSV data loaded, yet without invariant checks. This allows loading invalid data, which can then be processed and errors reasons can be given to the user. Finally, if the flag is true and errors are not empty, data will not contain the short rows identified.

CSV Java native implementation

The implementation of VDM native calls is done in Java. `CSVLib.java` file implements the VDM link, as well as various specialised VDMJ value-AST handling methods. These might be useful for other libraries requiring handling VDMJ values within VDM native method call implementations in Java. For example, we added for the value-AST in Java the VDM equivalent of record `mu`-expressions for record updates in Java. This made the VDM record construction and update process in Java much alike one would do in VDM itself.

Finally, `CSVLib.java` delegates to a separate `CSVParser` Java interface,which provides the necessary services needed by the native call implementations. This is important in order to separate the VDMJ value-AST processing from the low-level CSV parser implementations, hence making extending the library with other CSV parsers a relatively stratight forward process.

```
public interface CSVParser {
    public Iterator<String[]> parseCSV(final InputStream stream)
```

```
        throws IOException;
    public String lastError();
    public void clear();
    public CSVSettings getSettings(); }
```

Any low-level implementation only has to return a Java iterator view of the input stream for the `CSVLib.java` to work according to the design principles described here. Other methods are self-explanatory.

The separation between Java implementation of VDM native calls and other Java code is important because debugging of VDM native calls alongside VDM specification is relatively tricky to setup. Such considerations and difficulties have now been resolved and can be reused by other VDM library developers that require VDM native call implementations.

In practice, the setup enables the library developer to use VDMJ's command-line debugger to handle VDM library specification debugging, whilst using VSCode Java debugger to tackle Java's implementation of VDM native library calls. In future, we want to integrate the debugging environment of VDM native library calls, such that we use VDM VSCode DAP protocol [**?**]. This way, the VDM library developer can handle VSCode debugging of both VDM library specification and examples, alongside VSCode Java debugging.

CSV data and errors

The CSV data type is contains CSV settings, headers, and data matrix as a sequence of sequence of values. In `CSVLib`, an error occurs when one of the invariant check fails. A CSV error records cell (row and column) position alongside a non-empty explanation as to why the error has occurred.

```
  Data :: settings: CSVSettings headers: Headers matrix: Matrix;
 Error :: rowNo: nat1 colNo: nat1 reason: Reason;
```

After a successful call to `csv_read_data`, users can inspect what invariant has been violated by calling the function:

```
    csv_invariants_failed: Data -> set of Error
```

CSV state and operations

CSV native calls and error handling are expressive and accessible through functions and types described above. Nevertheless, their expressivity can lead to involved specifications. To avoid this for end users, we provide a state-based interface with operations that give access to the CSV data matrix, any errors found, as well as other (IO) or usability errors.

```
    state CSV of
        file  : [Path]   parser: CSVParser       ferr: Reason
```

```
     strict: bool      pos    : set of Errors   data: Data
  inv mk_CSV(file, -, -, strict, pos, data) ==
     (file <> nil => file_status(file) = <Valid>) and
     (strict => csv_invariants_failed(data) = pos = {})...
```

The CSV state contains the CSV file path and parser kind, low-level (IO and other non-CSV) error reasons, strict invariant validation, positions of invariant errors, and CSV data matrix. We show some of the state invariants, which say that non-nil file status must be valid (*e.g.* exist, not be a directory, *etc.*), and `strict` CSV does not tolerate any errors (including short row IO errors).

There are three operations for end users. The `loadTrivialCSV` example above simply calls `loadCSV` with an appropriate header created. `loadCSV` ensures the file path is valid, (re-)sets up all relevant state parameters and loads the CSV data. Data loading follows the logic explained for native call `csv_read_data`, where various cases are given descriptive error and/or information messages to the user. At this stage, both CSV data and invariant check errors are available as part of the state for inspection and further processing.

```
loadTrivialCSV: Path * CSVParser * seq1 of CSVType * bool ==> ();
loadCSV       : Path * CSVParser * CSVSettings * Headers * bool ==> ();
printCSV      : Path ==> ();
```

Finally, `printCSV` prints out the loaded CSV to the given file path. This presumes loading has taken place successfully, and that the given path is not the same as the CSV file path itself to avoid data overwriting.

5 Examples

Next ,we illustrate the intended use of our library through a few examples. First, users have to provide information about CSV headers. In our example, we have a list of names with their corresponding age, weight (in kg), height (in cm), and BMI. We then create the corresponding VDM header as:

```
EXAMPLE_HEADERS : Headers =
      [mk_Header("Name"      , <String> ,  "Name", nil, COL_INV_UNIQUE_NAME),
       mk_Header("Age"       , <Integer>,       MIN_AGE, CELL_INV_AGE, nil),
       mk_Header("Weight(Kg)", <Float>  , MIN_WEIGHT_KG, CELL_INV_WEIGHT, nil),
       mk_Header("Height(cm)", <Float>  , MIN_HEIGHT_CM, CELL_INV_HEIGHT, nil),
       mk_Header("BMI"       , <Float>  ,       MIN_BMI, CELL_INV_BMI, nil) ];
```

Default values are provided to test for CSV comma-sequences (*e.g.* CSV cells without any value), alongside cell and column invariant examples. These default values are used to ensure the implicit row invariant check that row size musts match header size. The cell invariants on age, weight, height and the column invariant on uniqueness are similar to the one showed above (see **??**).

Next, we want to define a row invariant that checks the redundant BMI fields are consistent with respect to height and weight. Arguably the CSV should not have a BMI field. Having said that, many CSV files do contain such redundant information [**?**], which are frequently not accurate with respect to intended values. The BMI check row invariant is defined as:

```
(lambda h: Headers0, r: Row &
    if 5 > len h then "invalid BMI header"
    else
      (let bmi: real = calculated_bmi(r) in
        if approx_eq(r(5), bmi, PRECISION) then nil
        else "invalid BMI for given CSV weight and height"));
```

For the given row, the user-defined function `calculate_bmi` calculates the BMI using the row values for height and weight, wheread `r(5)` gets the CSV BMI value, which needs to be approximately equal to the calculated value. Approximation here considers a particular number of digits precision for comparison between the cell value and the calculated one.

Other examples are provided in `CSVLib.vdmsl` distribution (`CSVExample.vdmsl` in [**?**]). They show CSV parsing with different IO-parsers, CSV printing, escaped quotes in string cells (*e.g.* strings across multiple lines), default values (*e.g.* comma-sequences in CSV row), short rows (*e.g.* rows smaller than expected header), various types of invariant violation (*e.g.* implicit and user-defined), and so on.

6 Results and discussion

CSV processing is an important part of the computation involving multiple application domains. The area is plagued with subtle errors and inconsistencies, which would ultimately invalidate the systems associated with the data. When that represents system architecture itself (*e.g.* dialyser state machine representation [**?**]), as opposed to system data, consequences can be catastrophic.

In this paper, we presented a formally defined CSV library in VDM that adheres to our **SAFE** design principles (**??**). The library architecture (**??**) is **S**imple, given its layered access to functionality and ultimately near-trivial user-interface access points. It is also **A**ccurate, given the presence of multiple kinds of user-defined invariant and other structural and data validation checks, such as detection of short rows and cell value type consistency with respect to declared headers. It is **F**ast, as the architecture layout allows for plug-and-play of different CSV-IO parsing, and we implemented a fast one and choose another with the fastest available benchmarks. Finally, it is **E**ffective, given its combination of speed, ease of use, multiple capabilities around CSV format handling, CSV settings, errors handling, and so on.

Acknowledgements We acknowledge XYZ funding. We very much appreciate fruitiful discussion with Aaron Buhagiar about CSV parsing, and Nick Battle about VDMJ native calls setup.