Specification-based CSV support in VDM

Leo Freitas and Aaron John Buhagiar

School of Computing, Newcastle University,
{leo.freitas, A.J.Buhagiar2@}@newcastle.ac.uk

Abstract. CSV is a widely used format for data representing systems control, information exchange and processing, logging, *etc*. Nevertheless, the format is riddled with tricky corner cases and inconsistencies, which can lead 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, which also includes a translation and proof environment for VDM in Isabelle.

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 in organ transplant allocation[1], embedded systems representation as in state machines for medical devices [3, 12], pharmaceutical applications [10], log files on various kinds of systems, databases, payments, government systems and so on. Despite there being a standard (RFC 4180) [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 [5], XML and various spreadsheet formats.

Therefore, we argue it is important to have a formal underpinning to CSV-file format, with enough flexibility and variation to cater for realistic applications. For that, we choose VDM.

The Vienna Development Method (VDM) has has been widely used both in industrial contexts and academic ones covering several domains, such as Security [6, 7], Fault-Tolerance [11], Medical Devices [8], 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.

We decided to depart from the standard VDM-CSV library because it is slow, cannot handle various CSV dialects, relies on CSV positioning management (*e.g.* knowing

[1] https://unos.org/data/
[2] https://www.rfc-editor.org/rfc/rfc4180
[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/

where in the file you are; reading one CSV line at a time), and has no checks on the CSV data itself.

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.

2 Background

CSV is plagued with a variety of fiendish scenarios leading to unexpected errors [1, 10, 12]. 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 as column ordering or row data redundancy consistency (*e.g.* weight, height and BMI info must be correlated). Finally, also an overall file invariant. 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://github.com/overturetool/vdm-vscode), as well as VDMJ [2].

Related to formal CSV (and data sources) formal processing, we worked on generation of VDM files from XSD files describing the data dictionary of the EMV payment systems [4], and similar efforts have been applied to the Function Mockup Unit standard [6]. Another example is for loading the control system finite state machine definitions in CSV for a dialyser [9] and a brain pace maker [12]. These read in data in the corresponding (CSV or XML) format(s), and generate a bespoke VDM model corresponding to implicitly understood design decisions.

These applications of CSV to VDM are different from our approach here, which focuses on raw CSV data loaded within a VDM data type that can then be further checked for various structural consistencies through dynamic invariants. Similar to our `CSVLib.vdmsl` library is the one defined within VDMJ standard libraries (`CSV.vdmsl`): it provides users with native calls to a simple CSV parser, returning a sequence of wildcard (`?`) types.

3 Design Principles

Our VDM CSV library has four core design principles:

1. **S**imple: ease of use is crucial, given CSV processing is a pervasive 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;

[6] https://github.com/INTO-CPS-Association/FMI-VDM-Model

3. **F**ast: CSV input can often be large; hence, varied and computationally efficient IO parsing solutions are important [7];
4. **E**ffective: there are multiple CSV format-variants [8]; hence, format variations are important.

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

4 Library architecture

Next, we present how the library architecture that implements our design principles (see Figure 1). 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, and so on.

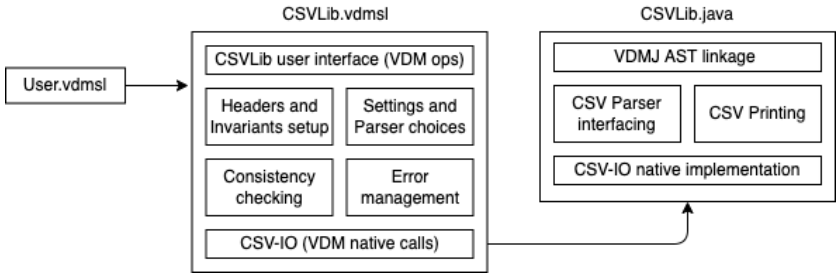

Fig. 1: `CSVLib.vdmsl` architecture.

`CSVLib.vdmsl` provides various functionalities including:

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

[7] https://github.com/uniVocity/csv-parsers-comparison
[8] https://commons.apache.org/proper/commons-csv/

`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 the CSV functionality, whereas the later provides the most convenient access. For instance, the simplest use of the CSV library is a call to the operation:

```
loadSimpleHeadersCSV(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 `uniVocity` CSV, 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, whilst 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 in a row. 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, the header below states that age is an integer with 18 as its default value

```
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 four CSV invariants, which are captured by the following VDM function types:

```
CSVCellInv = (CSVType * CSVValue -> Reason);
CSVRowInv  = (Headers * Row -> Reason);
CSVColInv  = (Header  * TransposedRow -> Reason);
CSVFileInv = (Headers * Matrix -> 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 the 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 column of cells to be checked, viewed as a transposed row. Finally, a file invariant defines properties across the whole CSV matrix.

Some invariant kinds have an implicit check, regardless of whether a user-defined invariant is given or not. For cells, this checks the values against their corresponding declared header types, whereas for rows, we implicitly check that the header size corresponds to row size (*e.g.* no short rows); there are no implicit column or file 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 &
6       if card elems c <> len c then "no duplicate ages are allowed" else nil);
```

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; 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 the Java CSV parsers. It expects a valid path, a known parser, settings and valid headers; it returns a tripple. In the result, if the success flag is **`false`**, some unexpected Java error occurred that can be inspected by a call to `lastError()`; the other two parts of the tripple representing short-row errors and data are empty. Otherwise, if the success flag is **`true`** and errors are not empty, then CSV parsing has identified short rows (*e.g.* row's size smaller than header's size), and they contain details where such errors occured in the CSV file. If the success 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 success 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 VDM-record updates. This made the VDM record construction and update process in Java much like one would do in VDM itself. For instance, the `loadSimpleHeadersCSV` call above does not provide a header name. This will come from the CSV itself, and must update the header name field akin to `mu(header, name |-> "Age")`. In Java, this is performed as

```
namedHeaderAtI = ValueFactoryHelper.muRecord(headerAtI,
    ValueFactoryHelper.mkFieldMap(
        Arrays.asList(HEADER_FIELD_NAME),
        ValueFactoryHelper.mkValueList(
            ValueFactoryHelper.mkString(nameStr)),
        Arrays.asList(true), ctx
    ), ctx);
```

The `ValueFactoryHelper` class provides a number of useful VDM value-AST support methods. The one above performs the record update on the field name given with the value passed. This illustrates how the `CSVLib` library can also be used by other libraries wanting to perofrm complex VDM value-AST manipulations.

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-IO parser implementations. This makes extending the library with other CSV parsers a relatively stratight forward process: users can implement the interface with their preferred CSV parser, and add the corresponding jar-file to the VDM-VSCode classpath.

```
public interface CSVParser {
    public Iterator<String[]> parseCSV(final Reader reader)
        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. We use Java `Reader` as the preferred input type in order to take file encoding issues into account.

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.

CSV data and errors

The CSV data type 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 position (*i.e.*, row and column) 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 invariants have been violated by calling the function

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

It returns the set of errors with the (row/col) position and a reason why the invariants have failed.

CSV state and operations

CSV VDM 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). The state and operations effectively illustrate how the VDM native calls can be used.

There are three operations for end users. The `loadSimpleHeadersCSV` operation example above calls `loadCSV` with an appropriate headers created. The `loadCSV` operation 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 VDM 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.

```
loadSimpleHeadersCSV: Path * CSVParser * seq1 of CSVType * bool ==> ();
```

```
loadCSV       : Path * CSVParser * CSVSettings * Headers * bool ==> ();
printCSV      : Path ==> ();
```

Finally, the operation `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, and a corresponding CSV file will have been created (or overwrriten).

5 Evaluation

The high-performance of our SAFE-CSV library is a core design principle. To demonstrate this, several CSV files, ranging from 100 to $35,000$ rows were used to compare the performance of our library against the standard CSV library in VDM. The VDM CSV library provides limited features in terms of the variety of formats it can accept for the CSV input, its low-level IO mechanism (*i.e.,* read line by line) and the fact there are no data validation capabilities available. The scope of the performance testing was limited to largest subset of features allowed by the standard CSV library. SAFE-CSV provides a richer approach to data validation than VDM CSV; however, these features were not used for this section to maintain parity between the two libraries during the performance test.

As SAFE-CSV also has the ability to use multiple different parsers, the library was tested using its Native parser and, the faster, Univocity parser[9].

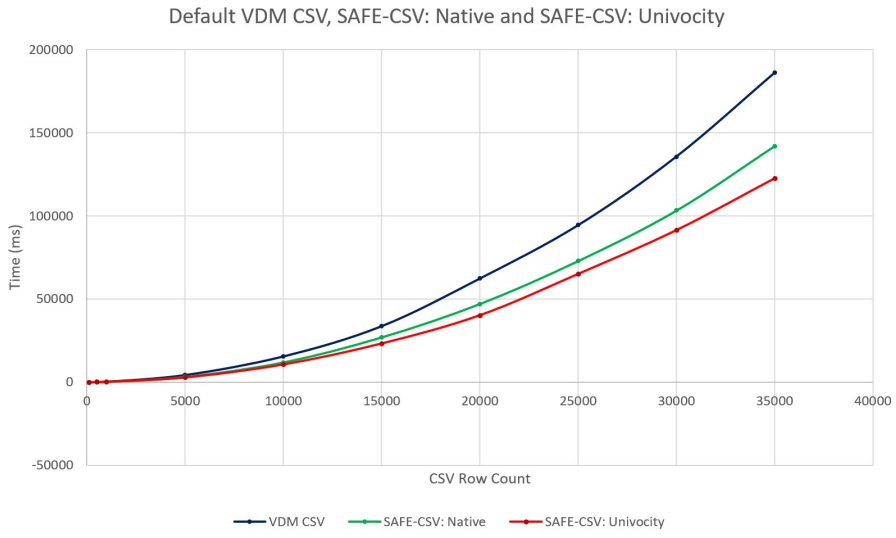

Fig. 2: VDM CSV and SAFE-CSV library performance comparison

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

Figure 2 shows the performance of the libraries using the same input data. The SAFE-CSV parsers performed significantly better than the standard library showing a speed-up of ~ 1.5x. The Univocity parser performed even better with larger datsets averaging a higher speed-up rate even compared to the Native parser.

6 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 Listing 1.1).

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 len h < 5 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, whereas `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 of precision for comparison between the cell value and the calculated one.

Other examples are provided in `CSVLib.vdmsl` and `CSVExample.vdmsl`[10]. 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.

[10] https://github.com/leouk/VDM_Toolkit/tree/main/plugins/vdmlib

7 Results and discussion

In this paper, we presented a formally defined CSV library in VDM that adheres to our **SAFE** design principles (in Section 3). The library architecture (Section 4) 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 invariants 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. 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.

Compared to the standard VDM CSV library, our SAFE-CSV performed better, yielding over a 50% increase in speed. Whilst the more robust invariant system of the SAFE-CSV library was not used for performance analysis, the infrastructure for its execution and in-built type-checking are still present throughout. The performance impact of invariant checking is negligible. It is comparable to users encoding such check inv VDM themselves, on top of the standard VDM CSV library for IO anyhow.

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 [3]), as opposed to system data, consequences can be catastrophic. For system architecture CSV files, tools like VDMJ's `CSVReader` or bespoke tools like those in [4, 9, 12] are suited; whereas for general data manipuation, varied CSV dialects, and IO speed, `CSVLib` library is better suited.

A key motivation for this work was the need for a more robust data import functionality to analyse data produced by an organ preservation medical device. The standard VDM library proved to be too limited in the CSV format it can handle. Additionally, it provides no data validation capabilities, which are necessary for safety critical applications.

Future Work. In future, we want to integrate the debugging environment of VDM native library calls, such that we use VDM VSCode DAP protocol [13]. Moreover, on the CSV format itself, nested CSV formats, further CSV types, multiple CSV headers per file, *etc*. This way, the VDM library developer can handle VSCode debugging of both VDM library specification and examples, alongside VSCode Java debugging.

Acknowledgements We acknowledge NCSC funding on development of safer payment protocols. We very much appreciate fruitful discussion with Nick Battle about VDMJ native calls setup.

References

1. Abba, A.H., Hassan, M.: Design and implementation of a csv validation system. In: Proceedings of the 3rd International Conference on Applications in Information Technology. pp. 111–116. ICAIT'2018, Association for Computing Machinery, New York, NY, USA (2018), https://doi.org/10.1145/3274856.3274879

2. Battle, N.: VDMJ User Guide. Tech. rep., Fujitsu Services Ltd., UK (2009)
3. D.Harrison, M., Freitas, L., Drinnan, M., Campos, J.C., Masci, P., di Maria, C., Whitaker, M.: Formal Techniques in the Safety Analysis of Software Components of a new Dialysis Machine. Science of Computer Programming 175, 17–34 (Feb 2019)
4. Freitas, L.: Vdm at large: Modelling the emv 2nd generation kernel. In: 21st Brazilian Symposium in Formal Methods, SBMF. pp. 26–30 (2018)
5. ISO: Iso/iec 21778 information technology — the json data interchange syntax. Tech. rep., ISO (2017)
6. Kulik, T., Macedo, H.D., Talasila, P., Larsen, P.G.: Modelling the HUBCAP Sandbox Architecture In VDM – a Study In Security. In: Fitzgerald, J.S., Oda, T., Macedo, H.D. (eds.) Proceedings of the 18th International Overture Workshop. pp. 20–34. Overture (December 2020)
7. Kulik, T., Talasila, P., Greco, P., Veneziano, G., Marguglio, A., Sutton, L.F., Larsen, P.G., Macedo, H.D.: Extending the formal security analysis of the hubcap sandbox. In: Macedo, H.D., Thule, C., Pierce, K. (eds.) Proceedings of the 19th International Overture Workshop. Overture (10 2021)
8. Macedo, H.D., Larsen, P.G., Fitzgerald, J.: Incremental Development of a Distributed Real-Time Model of a Cardiac Pacing System using VDM. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008: Formal Methods, 15th International Symposium on Formal Methods. Lecture Notes in Computer Science, vol. 5014, pp. 181–197. Springer-Verlag (2008)
9. Martinkute, E.: Formal Specification of a Novel Infant Dialyser. Master's thesis, School of Computing Science, Newcastle University UK (Aug 2016)
10. Nazario, A.C.: Computer System Validation CSV in Data Integrity Implementation Strategies for Pharmaceutical Industry. Master's thesis, Industrial and Systems Engineering Department (2018)
11. Nilsson, R., Lausdahl, K., Macedo, H.D., Larsen, P.G.: Transforming an industrial case study from VDM++ to VDM-SL. In: Pierce, K., Verhoef, M. (eds.) The 16th Overture Workshop. pp. 107–122. Newcastle University, School of Computing, Oxford (July 2018), TR-1524
12. Pollitt, A.: Verifying the CANDO Project Optrode Command Interface in eCv. Master's thesis, School of Computing Science, Newcastle University UK (August 2018)
13. Rask, J.K., Madsen, F.P., Battle, N., Freitas, L., Macedo, H.D., Larsen, P.G.: Advanced vdm support in visual studio code. In: Proceedings of the 20th International Overture Workshop. pp. 35–50 (2022)