Implementing CPSLint: A Data Validation and Sanitisation Tool for Industrial Cyber-Physical Systems

Uraz Odyurt[1], Ömer Sayilir[2], Mariëlle Stoelinga[2] and Vadim Zaytsev[2]

[1]Dynamics Based Maintenance, ET Faculty, University of Twente, The Netherlands
[2]Formal Methods and Tools, EEMCS Faculty, University of Twente, The Netherlands

Abstract

Raw datasets are often too large and unstructured to work with directly, and require a data preparation phase. The domain of industrial Cyber-Physical Systems (CPSs) is no exception, as raw data typically consists of large time-series data collections that log the system's status at regular time intervals. The processing of such raw data is often carried out using ad hoc, case-specific, one-off Python scripts, often neglecting aspects of readability, reusability, and maintainability. In practice, this can cause professionals such as data scientists to write similar data preparation scripts for each case, requiring them to do much repetitive work. We introduce *CPSLint*, a Domain-Specific Language (DSL) designed to support the data preparation process for industrial CPS. CPSLint raises the level of abstraction to the point where both data scientists and domain experts can perform the data preparation task. We leverage the fact that many raw data collections in the industrial CPS domain require similar actions to render them suitable for data-centric workflows. In our DSL one can express the data preparation process in just a few lines of code. *CPSLint* is a publicly available tool applicable for any case involving time-series data collections in need of sanitisation.

Keywords

Domain-Specific Language, Data validation, Data sanitisation, Industrial CPS, Tool support

1. Introduction

Data from Cyber-Physical Systems (CPS) is crucial for gaining insight into the functioning of these complex systems, both in the form of real-time fault detection and postmortem analysis of failures. Examples of CPS are high-tech production machinery, e.g., semiconductor photolithography machines, die bonder machines, and so forth. Runtime traces from these systems can often contain corrupted data originating from, for example, faulty sensor readings, missing readings due to sensor outages, or even human error while managing this data. When performing diagnostics with CPS data, it is important to sanitise the data and ensure such corruptions would not interfere with the intended data processing. In the context of industrial CPS, data sanitisation requires both domain knowledge and programming skill, thus often requiring both a domain expert and a data scientist to be involved in performing the task. Besides the fact that it requires multiple domains of expertise, the code written to sanitise a specific dataset is often not re-applicable to other cases, creating a situation in which time and effort are spent writing single-use Python scripts.

Ideally speaking, such data sanitisation would be designed best if it is done by a domain expert who possesses programming skills as well. As this is seldom the case, having a simplified and contained language instead of a general programming language, e.g., Python, would enable domain experts for this task. Domain-Specific Languages (DSLs) are a good fit when it comes to context-specific language design.

Following this rationale, we introduce CPSLint, a DSL built to aid the data sanitisation process. The language raises the level of abstraction for performing data sanitisation, making it accessible to domain experts and allowing them to perform this task without requiring substantial programming knowledge. It also has a relatively compact declarative syntax, making it also a suitable tool for data scientists by significantly reducing the amount of code they would need to write to sanitise a dataset.

Figure 1 depicts an example of a typical CPSLint workflow. The compiler takes a CPSLint specification and a raw CSV file as input. Our implementation then uses these to generate a human-readable Python script, capable of sanitising the raw CSV according to the provided definition. This script can then be run using a Python interpreter to obtain a sanitised CSV.

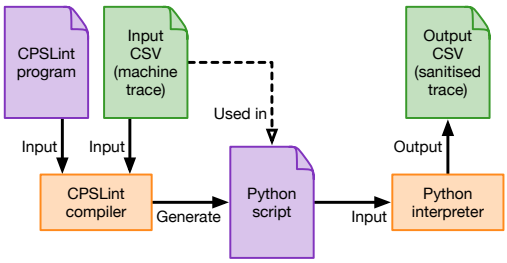


Figure 1: A typical CPSLint workflow for sanitising industrial CPS machine traces.

Cyber-physical systems are complex systems combining cyber elements, i.e., computational and networked components, with interactions in the physical domain. Their design often integrates multitudes of subsystems, working in tandem to perform specific, specialised tasks. Thus, CPS are purpose-built machines. The industrial variants exhibit a significantly higher level of complexity and precision, excelling in high-yield production for high-tech industries. It is standard practice to constantly monitor industrial CPS for machine behaviour tracking and fault detection/prevention. Time-series data collection is the de facto monitoring data format for these devices.

Domain-specific languages are software languages with a higher level of abstraction than General-purpose Programming Languages (GPLs) [1]. They are designed for

Post-proceedings of BENEVOL 2025
u.odyurt@utwente.nl (U. Odyurt); o.f.sayilir@utwente.nl (Ö. Sayilir); m.i.a.stoelinga@utwente.nl (M. Stoelinga); vadim@grammarware.net (V. Zaytsev)
0000-0003-1094-0234 (U. Odyurt); 0009-0009-8860-2316 (Ö. Sayilir); 0000-0001-6793-8165 (M. Stoelinga); 0000-0001-7764-4224 (V. Zaytsev)

specific tasks and enable domain experts to perform programming using familiar terms and concepts. The output of DSLs need not be executable, as in the case of LaTeX, YAML, and HTML, which are all markup languages. In short, DSLs make it easier for non-programmer domain experts to perform tasks programmatically within their domain than GPLs do.

Contribution We showcase a DSL that was built based on similarities across the data validation and sanitisation domains, combined with information from the cyber-physical domain. The current implementation of CPSLint provides a small but powerful declarative data sanitisation language. The compactness of our current language implementation is demonstrated by a relatively small codebase, i.e., 1978 lines of code. The data involved in our demonstrations, alongside the CPSLint code, are openly available [2][1].

The theoretical background leading to CPSLint as a tool and more detailed information on use-cases, alongside systematic benchmarking of computational performance for different modes, is provided in the SLE 2026 paper [3].

2. Data

While it can be adapted in other scenarios, CPSLint is primarily designed to work with industrial CPS data. The term 'data' in this context loosely refers to the combination of sensor data, alongside contextual logs. Accordingly, sensor data is recorded as time-series, covering one or more metrics. While the recoding is commonly done at a high frequency, e.g., 1 kHz, each metric will be recorded with a frequency supported by the respective sensor and collection logic. As an example relevant to our demonstration use-case, power metrics such as voltage and current can be collected from different components on a system. Depending on the desired granularity or available support, such collections could happen at the component level, the controller level (for instance motor controller), at the subsystem level, or at the system level.

The contextual logs are expected to contain the signalling information related to a system's activity, i.e., start and stop signals per task or subtask. While being a common practice, the granularity at which such signals are recorded depends on the designer's foresight. It must be mentioned that given access to the underlying software element of an industrial CPS, it is relatively easy to extend contextual logging.

2.1. Data format

When considering time-series data, the de facto format is Comma-Separated Values (CSV) files. The *timestamp* column is almost always present, with rare exceptions that consider an ordered index instead. One or more metric columns are included as well, depending on the modality of collected data. For instance, CSV files from our demonstration use-case include voltage, current and energy readings, as detailed below:

- Timestamps (S) column with values in ms
- Voltage (V) column with values recorded at 1 kHz
- Current (A) column with values recorded at 4 kHz
- Energy (J) column with values reflecting consumed energy
- Universal Asynchronous Receiver-Transmitter (UART) messages as string data

The UART message column is populated sparsely, only indicating subtask starts and stops, or transmitting information such as processor core temperature.

2.2. Data compartmentalisation

One of the preprocessing steps governing the dataset formation for ML model training is the compartmentalisation of data according to execution phases. Execution phases are segments of monitoring data, reflecting the behaviour of the system under scrutiny per individual task or subtask during an execution. In other words, the time-series data from the start till the stop signal is associated with the respective task. In this fashion, the data can be *cut* in individual phases. Figure 2 depicts the concept and provides the phases considered in our demonstration use-case.

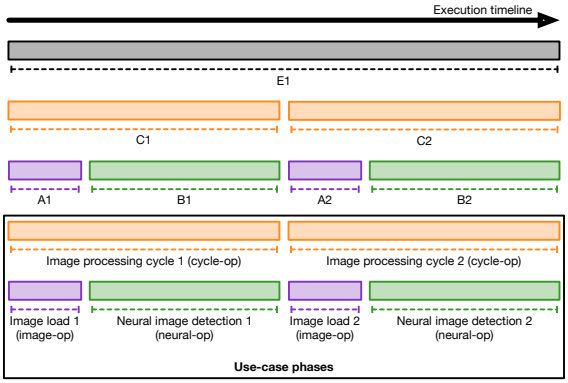


Figure 2: Different data compartmentalisation granularities within an execution timeline, visualising repeated phase types during consecutive rounds of tasks/sub-tasks, plus the phases considered for processing consecutive image data.

2.3. Emulating data corruptions

CPSLint is capable of both sanitising and compartmentalising time-series data. To be able to systematically evaluate and demonstrate these functionalities, a series of corrupt data examples are devised. This is done through Python scripts manipulating reference data, i.e., correct data, and emulating known corruption patterns. The considered reference data is collected from an embedded platform with repetitive sequential activity, as described by Odyurt et al. [4].

The corruptions are applied to the reference data in non-overlapping blocks of rows, in this case 10-row blocks. The total corruption rate is 0.5%, which is a good amount considering how extensiveness the reference data is. Note that while not affecting CPSLint functionality, the block size and corruption rate are configurable parameters. Blocks are selected at random unless the emulated corruption is targeted. Corruptions can be applied individually, or in combination. The following types are supported:

Data type mismatch Numeric fields are injected with invalid characters, e.g., letters or special symbols, rendering them non-parsable. In some cases, the Universal Asynchronous Receiver/Transmitter (UART) identifier field is also corrupted, simulating noisy channel logs.

[1]Also available at: https://github.com/omersayilir75/CPSLint

Data type mismatch with targeted UART Similar to the previous corruption, but biased towards affecting rows associated with a specified UART identifier. This enables testing of selective corruption of device-specific data. UART data fields are especially important for steering the targeted solution and as such, it is important to have cases with noisy UART messages.

Out-of-bounds values Numeric fields are replaced with extreme constants, e.g., "99 999.999", that are far outside expected measurement ranges. Depending on configuration, this may affect all numeric columns or a random subset within each block.

Out-of-order rows with reliable timestamps Rows in a block are randomly permuted, but their original timestamps are preserved. This emulates reordering of messages while maintaining temporal information.

Out-of-order rows with unreliable timestamps Rows are shuffled and new, monotonically increasing timestamps are generated for the block. This emulates cases where both order and timing data are compromised during collection.

Missing fields A randomly chosen numeric column is blanked out across all rows in a block, representing systematic loss of a sensor channel.

Missing rows Entire blocks of rows are deleted, optionally biased toward containing a specified UART identifier. This models partial loss of data segments, such as dropped packets or truncated logs.

Misplaced end-of-line markers Rows within a block are concatenated with their successors, simulating the effect of missing or corrupted line delimiters. This results in malformed records and broken row alignment, resembling common parsing errors in raw log files.

3. Using CPSLint

3.1. Setup

To set up CPSLint in a workspace, the DSL needs to be configured with a set of parameters. These include location of the input CSV files, the desired output folder, a command to invoke Python, and which pipeline to use for execution of the specification. The parameters are to be defined in a YAML file in the folder where the CPSLint definitions are located. An example of such a configuration file is given in Listing 1.

```
inputFolder : ~/data/input
outputFolder : ~/data/output
pythonCommand : python
runWith: interpreter # Current options:
    interpreter, compiler
```

Listing 1: The config.yaml

As can be seen in Listing 1, CPSLint allows for the use of multiple back-ends: a compiler pipeline, meant for generating well performing idiomatic Python code, and an interpreter pipeline, a back-end meant for debugging and diagnostic purposes at the cost of performance. A more detailed overview of these pipelines is given in Section 4.

Regardless of which back-end pipeline is chosen, the user can use the language in one of two ways: through the Rascal REPL, or through the Language Server Protocol (LSP) interface provided by Rascal. Accessing CPSLint through the Rascal REPL allows the use of the language without the overhead of an IDE. When considering this option, the `Runner` module can be imported to run the CPSLint programs through calling the `run` function from the REPL, by giving the location of the program to be executed as an argument. An example of this usage pattern is given in Listing 2.

```
rascal>import Runner;
ok
rascal>run(|project://cps/Input/scripts/input.cps|);
```

Listing 2: An example showing how to run CPSLint specifications through the Rascal REPL.

Using the LSP implementation gives the user Visual Studio Code support, bringing CPSLint language features to the editor. These features can be enabled by importing the `CPSLintLanguageServer` module through the Rascal REPL and running the `main` function, which in turn registers the CPSLint language and makes the editor ready for use. In Figure 3, we illustrate what the IDE environment looks like. As can be seen, we provide syntax highlighting for the language and a way to run the current program.

3.2. Main features

The current implementation of the CPSLint language provides the following core feature set:

Data inspection and script generation This feature is meant as a starting point when working with a new dataset using CPSLint. Here the user can write a one-line command such as `inspect csv from 'input.csv';`, which instructs CPSLint to produce a specification based on the given CSV file. Users can then use this to continue defining how the CSV in question should be processed.

Data corruption remediation A large chunk of CPSLint's functionality revolves around the remediation of corrupt data. This includes features such as enforcing datatypes on columns to ensure that only values of an expected type are present. It can also set valid ranges for numeric columns, such that faulty readings are removed from the dataset. Additionally, it may filter unwanted substrings out of the data, ensuring that any unexpected characters that appear due to, for example, lossy channels are removed.

Data imputation CPSLint also allows for the imputation of missing data. The interpolation methods the language offers range from simple ones, such as using the mean or median of a column, or forward or backfilling the column with known values, to more involved ones, such as polynomial interpolation.

Data compartmentalisation CPSLint supports data compartmentalisation through the `export` command. Using

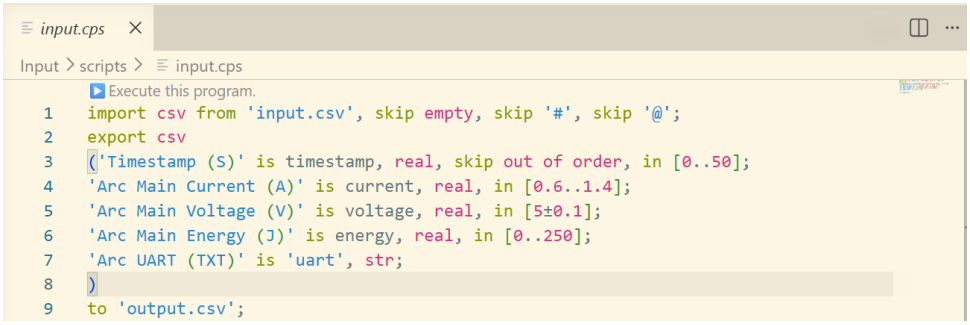


Figure 3: CPSLint language features running in Visual Studio Code.

this command, the user can specify based on which column they would like to split a file into smaller ones. For example,

```
export csv ... to 'output#.csv' cut when 'UART' is 'image loader'~;
```

instructs CPSLint to create multiple output CSV files, each covering a unique instance where image loading happens.

3.3. Example programs

Here, we will go over a few representative input programs for CPSLint and explain them in detail.

Example program 1: `out_of_bounds.cps` This program is representative of a case where it can be assumed that the input CSV has correct column types, but contains rows in which column values lie outside the defined valid ranges.

```
import csv from 'out_of_bounds.csv',
    skip empty;
export csv
(
  'Timestamp (S)' is timestamp, real,
    in (0..120];
  'Arc Main Current (A)' is current, real,
    in [0.4..1.0);
  'Arc Main Voltage (V)' is voltage, real,
    in [5±0.6];
  'Arc Main Energy (J)' is energy, real,
    in (0..400);
  'Arc UART (TXT)' is 'uart';
)
to 'out_of_bounds_fixed.csv';
```

Listing 3: CPSLint code for Example Program 1

In Listing 3, we show a program where first a CSV is imported through the `import` command, then empty lines are removed through the `skip empty` row filter; row filters are applied at the row level across the file. The second command in the file is the `export` command, here the columns found in the import file are mapped to rows in the output file through the `is` keyword. Then for all numeric columns, the DSL is instructed to view them as `real` values. Finally, the `in` keyword is used to indicate a valid numeric range for each column before exporting the CSV to the output file. Cells where the data is outside the defined valid range of a column are emptied without affecting the rest of the row.

When this program is run using the CPSLint compiler, the code in Listing 5 is generated as an intermediate artefact; when executed, this Python code produces the output CSV.

Example program 2: `datatype_mismatch.cps` The second program is representative to one that would be written when there are unexpected types encountered in columns where a certain type is expected. In this case there are string values in numeric columns that need to be removed.

```
import csv from 'datatype_mismatch.csv',
    skip empty, skip ~'#'~, skip ~'@'~,
    skip ~'$'~, skip ~'abc'~;
export csv
(
  'Timestamp (S)' is timestamp, real;
  'Arc Main Current (A)' is current, real,
    impute linear interpolation;
  'Arc Main Voltage (V)' is voltage, real,
    impute linear interpolation;
  'Arc Main Energy (J)' is energy, real,
    impute linear interpolation;
  'Arc UART (TXT)' is 'uart';
)
to 'datatype_mismatch_fixed.csv';
```

Listing 4: CPSLint code for Example Program 2

The `import` command in Listing 4 starts off similar to the one in the first example program, however here there are also file wide substring filters defined by the `skip` column filters in the command. The tildes in the `skip` filters signify if a string should be skipped, if it starts with a substring (tilde after the substring), if it ends with a substring (tilde before the substring), or if it contains the substring (substring between tildes). In other words, the tilde notation differentiates between substring filtering (if tildes are present) and exact string filtering (if tildes are absent). The `export` command starts off by enforcing the appropriate datatypes on the columns. Furthermore, the `export` command fills any cells that might have been emptied by previous operations, or were already empty in the input CSV, via the `impute` keyword, using `linear interpolation` as an imputation strategy.

The Python code generated by the CPSLint compiler can be found in Listing 7. It should be noted that most interpolation methods supported by Pandas rely on SciPy being installed.

3.4. Planned features

Future iterations of CPSLint are planned to bring features that would better serve the needs of the domain for which it is built. The declarative nature of the syntax of CPSLint makes the language relatively easy to extend at the syntax level, with new features requiring only minor edits of the grammar. The main additions would be to introduce the

```python
# Generated by CPSlint
# Input: out_of_bounds.cps

# Import libraries
...

# Auxiliary functions
...

def call_cpslint():
    # Read input CSV
    input_path = os.path.join("~/data/input", "out_of_bounds.csv")
    df_in = pd.read_csv(input_path)
    # Skip empty rows
    df_in = df_in.replace(r'^\s*$', pd.NA, regex=True).dropna(how="all")
    # Create output DataFrame
    df_out = pd.DataFrame()
    # Copy column "Timestamp (S)" as "timestamp" in the output DataFrame
    df_out["timestamp"] = df_in["Timestamp (S)"]
    # Copy column "Arc Main Current (A)" as "current" in the output DataFrame
    df_out["current"] = df_in["Arc Main Current (A)"]
    # Copy column "Arc Main Voltage (V)" as "voltage" in the output DataFrame
    df_out["voltage"] = df_in["Arc Main Voltage (V)"]
    # Copy column "Arc Main Energy (J)" as "energy" in the output DataFrame
    df_out["energy"] = df_in["Arc Main Energy (J)"]
    # Copy column "Arc UART (TXT)" as "uart" in the output DataFrame
    df_out["uart"] = df_in["Arc UART (TXT)"]
    # Cast values of timestamp to float if possible, set to NaN otherwise
    df_out["timestamp"] = pd.to_numeric(df_out["timestamp"], errors="coerce").astype(float)
    # Emptying cells in timestamp that are not in the range between 0. and 120., with inclusive set to right
    df_out.loc[~df_out["timestamp"].between(0., 120., inclusive="right"), "timestamp"] = np.nan
    # Cast values of current to float if possible, set to NaN otherwise
    df_out["current"] = pd.to_numeric(df_out["current"], errors="coerce").astype(float)
    # Emptying cells in current that are not in the range between 0.4 and 1.0, with inclusive set to left
    df_out.loc[~df_out["current"].between(0.4, 1.0, inclusive="left"), "current"] = np.nan
    # Cast values of voltage to float if possible, set to NaN otherwise
    df_out["voltage"] = pd.to_numeric(df_out["voltage"], errors="coerce").astype(float)
    # Emptying cells in voltage that are not in the range between 4.4 and 5.6, with inclusive set to both
    df_out.loc[~df_out["voltage"].between(4.4, 5.6, inclusive="both"), "voltage"] = np.nan
    # Cast values of energy to float if possible, set to NaN otherwise
    df_out["energy"] = pd.to_numeric(df_out["energy"], errors="coerce").astype(float)
    # Emptying cells in energy that are not in the range between 0. and 400., with inclusive set to neither
    df_out.loc[~df_out["energy"].between(0., 400., inclusive="neither"), "energy"] = np.nan
    # Write Output CSV
    output_path = os.path.join("~/data/output", "out_of_bounds_fixed.csv")
    df_out.to_csv(output_path, index=False)
call_cpslint()
```

Listing 5: Python code generated by running Example Program 1 using the CPSLint compiler

concept of domain-specific units (Volt, Watt, mAh) as native data types. This would enable us to perform safe and simple type conversions on columns. Another extension improving applicability, is to support more data source formats, e.g., HDF5 files or time-series databases. Future iterations of CPSLint could support compartmentalisation based on numerical columns, where, for instance, time-series data is cut according to peaks, valleys, and plateaus in these values. This is useful in cases where the data cannot be cut based on clear logged indicators due to their absence, unreliability, or corrupt state.

Another extension that would make the language more broadly applicable is the support for custom Python libraries within the compiler pipeline. These would contain functions taking either a `DataFrame` (for operations at the table level) or a `Series` (for operations at the column level) as their input. In this way, users can add their own case-specific algorithms to preprocess data using CPSLint. An example of how such an extension could look like is depicted in Listing 6.

```
using <library.py>;
import csv from 'input.csv', skip empty,
    skip '#', skip '@', perform
    <tableOperation>;
export csv
(
  ...
  'Arc Main Current (A)' is current, real,
    in [0.6..1.4], perform <rowOperation>;
  ...
)
to 'output.csv';
```

Listing 6: Syntax for using external libraries in CPSLint.

```python
# Generated by CPSlint
# Input: datatype_mismatch.cps

# Import libraries
...

# Auxiliary functions
...

def call_cpslint():
    # Read input CSV
    input_path = os.path.join("~/data/input", "datatype_mismatch.csv")
    df_in = pd.read_csv(input_path)
    # Blank out values that contain #
    df_in = df_in.replace(r'^.*' + re.escape("#") + r'.*$', '', regex=True)
    # Blank out values that contain @
    df_in = df_in.replace(r'^.*' + re.escape("@") + r'.*$', '', regex=True)
    # Blank out values that contain $
    df_in = df_in.replace(r'^.*' + re.escape("$") + r'.*$', '', regex=True)
    # Blank out values that contain abc
    df_in = df_in.replace(r'^.*' + re.escape("abc") + r'.*$', '', regex=True)
    # Skip empty rows
    df_in = df_in.replace(r'^\s*$', pd.NA, regex=True).dropna(how="all")
    # Create output DataFrame
    df_out = pd.DataFrame()
    # Copy column "Timestamp (S)" as "timestamp" in the output DataFrame
    df_out["timestamp"] = df_in["Timestamp (S)"]
    # Copy column "Arc Main Current (A)" as "current" in the output DataFrame
    df_out["current"] = df_in["Arc Main Current (A)"]
    # Copy column "Arc Main Voltage (V)" as "voltage" in the output DataFrame
    df_out["voltage"] = df_in["Arc Main Voltage (V)"]
    # Copy column "Arc Main Energy (J)" as "energy" in the output DataFrame
    df_out["energy"] = df_in["Arc Main Energy (J)"]
    # Copy column "Arc UART (TXT)" as "uart" in the output DataFrame
    df_out["uart"] = df_in["Arc UART (TXT)"]
    # Cast values of timestamp to float if possible, set to NaN otherwise
    df_out["timestamp"] = pd.to_numeric(df_out["timestamp"], errors="coerce").astype(float)
    # Cast values of current to float if possible, set to NaN otherwise
    df_out["current"] = pd.to_numeric(df_out["current"], errors="coerce").astype(float)
    # Impute with stategy: empty cells in current are replaced using interpolation with the
    linear method
    df_out["current"] = df_out["current"].interpolate(method="linear")
    # Cast values of voltage to float if possible, set to NaN otherwise
    df_out["voltage"] = pd.to_numeric(df_out["voltage"], errors="coerce").astype(float)
    # Impute with stategy: empty cells in voltage are replaced using interpolation with the
    linear method
    df_out["voltage"] = df_out["voltage"].interpolate(method="linear")
    # Cast values of energy to float if possible, set to NaN otherwise
    df_out["energy"] = pd.to_numeric(df_out["energy"], errors="coerce").astype(float)
    # Impute with stategy: empty cells in energy are replaced using interpolation with the
    linear method
    df_out["energy"] = df_out["energy"].interpolate(method="linear")
    # Write Output CSV
    output_path = os.path.join("~/data/output", "datatype_mismatch_fixed.csv")
    df_out.to_csv(output_path, index=False)
call_cpslint()
```

Listing 7: Python code generated by running Example Program 2 using the CPSLint compiler

4. Design and implementation

CPSLint is implemented using the Rascal meta programming language [5, 6], a "one-stop-shop" language workbench for rapid implementation of DSLs. The language offers its users a shared front-end and two back-ends: the *compiler*, which generates human-readable, idiomatic Python code that performs the tasks described by the CPSLint input program; and the direct interpreter in Rascal, providing granular insight into the data sanitisation process in the form of logs and intermediate CSV files. In this section, we go over the internals of CPSLint.

4.1. Language front-end

The syntax of CPSLint has been kept relatively simple, with the language only knowing three actions: `inspect` for initial analysis of the data, `import` for reading, and `export` for writing. The syntax of the language was created during design sessions with domain experts, with the goal of making a language that is familiar, sufficiently expressive, but not overly bloated. The entire abstract syntax definition, illustrating the simplicity of the DSL, is contained in Listing 8.

The parsing infrastructure is rather straightforward: we first parse the CPSLint programs as a concrete syntax tree, then implode it into a more workable abstract syntax tree.

```
module syntaxes::Abstract

alias Script = list[Action];
alias Mapping = tuple[str from, str to, list[Filter] filters];
alias Mappings = lrel[str from, str to, list[Filter] filters];
alias Border = tuple[str name, Stringish val];

data Action
    = ImportAction(DataFormat format, list[Filter] filters, str filename)
    | ExportAction(DataFormat format, list[Mapping] mappings, str filename, Border border)
    | InspectAction(DataFormat format, str filename)
    | Inaction(str message)
    ;

data DataFormat = CSV();

data Stringish
    = JustString(str text)
    | RegExp(str regex)
    | StartsWith(str begin)
    | EndsWith(str end)
    | Contains(str middle)
    ;

data Filter
    = SkipEmptyRows()
    | SkipEmptyColumns()
    | SkipOutOfOrder()
    | SkipIgnored(Stringish what)
    | InRange(real low, real high, bool includeLow, bool includeHigh)
    | EnforceType(SupportedType t)
    | EnforceOrder()
    | Impute(Strategy s)
    | IsCritical()
    | Unique()
    ;

data SupportedType = Natural() | Integer() | Real() | String() | Boolean() | UART();

data Strategy
    = Last()
    | Next()
    | Mean()
    | Median()
    | Constant(str c)
    | Interpolation(str method)
    ;
```

Listing 8: The abstract syntax definition of CPSLint.

The abstract syntax tree is then passed to the selected back-end for the current run.

4.2. Ad hoc Python sanitisation

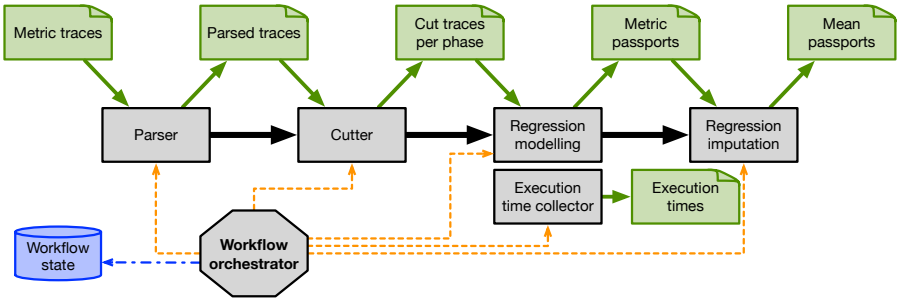
As both our inspiration and the actual use-case in which CPSLint is a suitable replacement for some of the stages, a Python project [7] containing multiple configurable workflows, composed of sequential pipeline stages, can be presented. Details of the monitoring data format and fields relevant to this use-case have been elaborated in Section 2. The project provides alternative anomaly detection and identification solutions for industrial CPS, based on traditional ML algorithms and CNN deep neural networks. Figure 4 covers three of these workflows, which prepare and process data for ML dataset formation. The follow-up workflows using these datasets and performing actual model training or inference, are not included, as there is no need for CPSLint's functionality at those stages.

In short, Figures 4a and 4b depict the data processing steps for processing data representing normal and anomalous system activity for traditional ML training. On the other hand, Figure 4c does the same, but for a CNN model design. As it can be seen, all three workflows involve *parsing* and *cutting* (compartmentalisation) of the data. These tasks match the functionality provided by CPSLint, i.e., machine trace ingestion, sanitisation, and compartmentalisation. As these steps are commonly present not only in these workflows, but in general most workflows processing industrial CPS data, the need for reusable tools such as CPSLint is apparent.

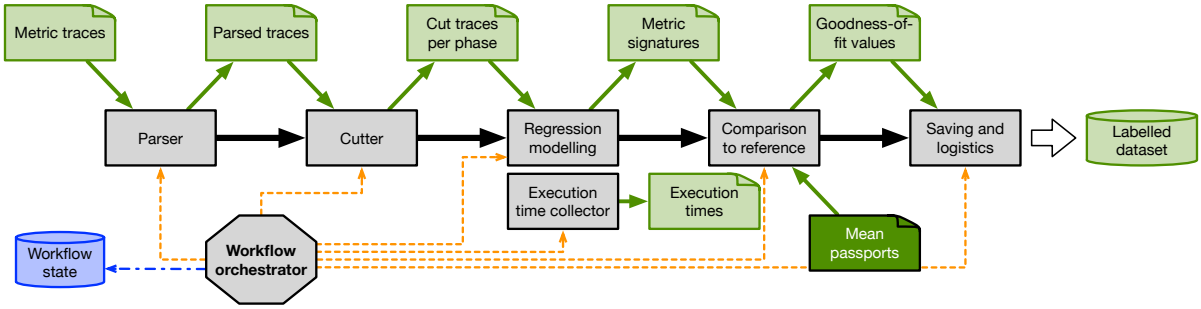
4.3. The compiler back-end

The compiler back-end takes the parsed abstract syntax tree and generates the corresponding Python code from it. This code is then run, producing the processed CSV file.

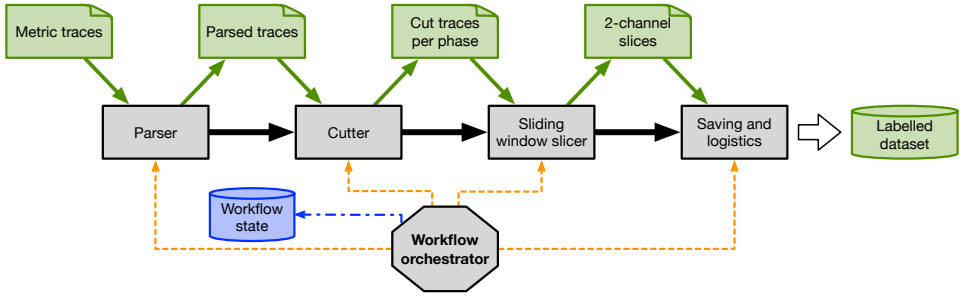
The compiler generates Python code that performs the operations described in the CPSLint input. The generated code leverages existing Python data science libraries such as Pandas, NumPy, and SciPy, enabling the compiler to produce high-performance, relatively compact code. Each code fragment generated by the compiler is preceded by a short



(a) Solution workflow processing normal system data, intended for traditional ML dataset formation.



(b) Solution workflow processing anomalous system data, intended for traditional ML dataset formation.



(c) Solution workflow processing system data, intended for CNN dataset formation.

Figure 4: Different workflows involving parsing and cutting of industrial CPS monitoring data: Traditional ML model training dataset formation flow handling a) "Normal" data and b) "Anomalous" data; c) CNN model training dataset formation flow. Note the presence of parsing and cutting in all such workflows as fundamental and repeating data processing stages.

description of its purpose. This can be seen in the example programs and their generated code shown in Section 3.3. Users can thus relate the generated code to the CPSLint specification and make changes to either the CPSLint program or the generated Python code if the compiled code does not match their expectations.

The pipeline for this back-end is illustrated in Figure 5. The figure starts with an `inspect` script, which generates a script describing the input dataset in its unprocessed form. The compiler generates a Python script that in turn generates a baseline CPSLint specification for the dataset. This is done with the help of the Pandas library, which is used to infer column names and types. The domain expert then refines the baseline script by adding how CPSLint should sanitise the dataset. This refined input is then compiled into a Python script, which is in turn run to output a sanitised version of the dataset.

4.4. The interpreter back-end

The interpreter back-end directly interprets the CPSLint code, performing the operations defined in the input using Rascal itself and thus forgoing compilation to Python entirely. This method of running CPSLint relies on our runtime library written in Rascal and thus cannot make use of the more mature, well-performing data science libraries available for Python, and therefore suffers a significant performance penalty. What this back-end does offer, however, is greater insight into the individual steps CPSLint takes while executing the input script. It does so by outputting an intermediate CSV after each operation, and by producing a timestamped log detailing the steps taken to perform the data sanitisation at runtime. An example of such a log file is given in Listing 9.

The tombstone diagram for using CPSLint with the interpreter back-end is shown in Figure 6. Like with the compiler back-end the process starts with providing CPSLint with an `inspect` script, which again generates a baseline CPSLint script. This back-end uses `lang::csv` from the Rascal standard library for type inference of the column names and types, thereby matching the functionality from the `inspect` implementation from the compiler. The domain expert again refines the baseline script and runs this through the interpreter, which in this case generates a sanitised CSV as its

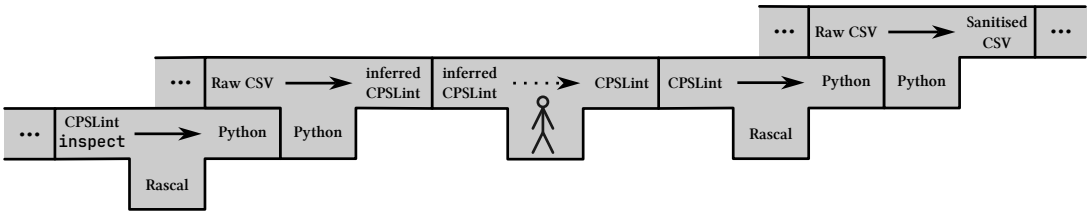


Figure 5: A tombstone diagram of the CPSLint compiler pipeline. Activities flow rightwards, with inspection and inferring of the data structure happening on the left side, followed by Python code generation to deliver the executable code operating on the actual machine trace. Ellipses indicate where the normal data processing flow connects. Note the role of a domain expert refining the inferred CPSLint specification.

```
2026-04-08 15:25:34.597 | [IO] Created intermediate CSV for Import action
    ~/data/output/intermediate/import_filters_applied.csv
2026-04-08 15:25:34.627 | [ImportFilter] Skipping empty rows
2026-04-08 15:25:37.019 | [RuntimeLib] Saved new intermediate csv after applying skipEmptyRows
    at ~/data/output/intermediate/import_filters_applied_skipEmptyRows.csv
2026-04-08 15:25:37.053 | [RuntimeLib] Updated
    ~/data/output/intermediate/import_filters_applied.csv after applying skipEmptyRows
2026-04-08 15:25:37.054 | [ImportFilter] Skipped empty rows
2026-04-08 15:25:37.055 | [IO] CSV used for Export:
    ~/data/output/intermediate/import_filters_applied.csv
...
2026-04-08 15:31:28.762 | [ColumnFilters] Enforced type real on column "energy"
2026-04-08 15:31:28.762 | [ColumnFilters] Emptying cells in energy that are not in the range
    between 0. and 400., with inclusive set to neither
2026-04-08 15:32:35.563 | [RuntimeLib] Wrote file where column "energy" only has values in range
    between 0. and 400., with inclusive set to neither to
    ~/data/output/intermediate/export_filters_applied_energy_range_applied.csv
2026-04-08 15:32:35.632 | [RuntimeLib] Updated
    ~/data/output/intermediate/export_filters_applied.csv after enforcing range (0. and 400.,
    inclusive: neither) on column "energy"
2026-04-08 15:32:35.633 | [ColumnFilters] emptied cells in energy that are not in the range
    between 0. and 400., with inclusive set to neither
2026-04-08 15:32:35.633 | [IO] Applied column filters
2026-04-08 15:32:35.640 | [IO] Saved final result to ~/data/output/out_of_bounds_fixed.csv
```

Listing 9: A truncated example of the log produced by the interpreter back-end

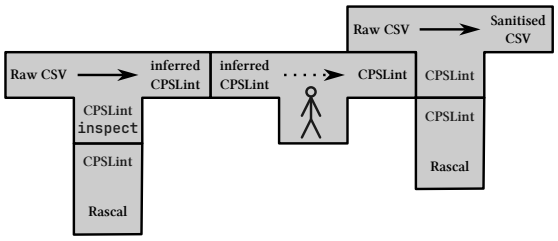


Figure 6: A tombstone diagram of the CPSLint interpreter: the presence of the interpreter (vertical block) of CPSLint written in Rascal allows us to see CPSLint specifications as transformations from raw CSV to its sanitised form.

main output. The interpreter writes the log alongside the main output, while creating intermediate files after each step the interpreter performs during the execution.

4.5. Integration into pipelines

Considering the flows from Figure 4, parsing and cutting stages in any of these, or any other time-series processing flow for that matter, can be replaced by the functionality provided by CPSLint. As described at the start of this section, CPSLint support two modes of operation: as a compiler and as an interpreter. Diagrams in Figure 7 present how the replacement and integration of CPSLint can be conceived.

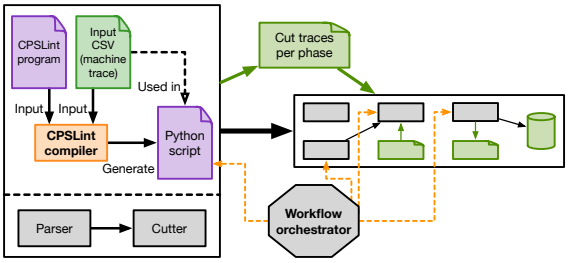
As described in Section 3, the interpreter mode is primarily intended for debugging and diagnostic purposes. The computational cost incurred and the I/O-heavy design will render this mode inadequate for production. Regardless, a systematic integration that can be driven with an orchestrator is shown in Figure 7b, mainly through a pseudoterminal (PTY) and a Read-Eval-Print Loop (REPL) interface.

The compiler mode on the other hand, could be seamlessly integrated with any workflow, as the orchestrator only has to drive Python scripts generated by CPSLint (Figure 7a).

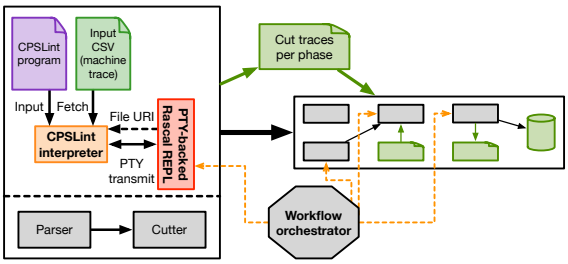
5. Related work

CPSLint was implemented in Rascal, which is just one of the language workbenches available today [8]. What is specific to Rascal is that it combines the role of meta-programming language and language workbench, with explicit support for source-code analysis and transformation [5]. Its design brings together grammar definition, algebraic data types, pattern matching, tree traversal, and source-location-aware manipulation in a single executable environment [6]. Among contemporary language workbenches, this places Rascal on the textual side, but with a particularly strong emphasis on programmable analyses and transformations [8].

Among the tools for data wrangling, sanitisation and



(a) CPSLint flow in compiler mode.



(b) CPSLint flow in interpreter mode.

Figure 7: Integration options considering the two modes of operation offered by CPSLint, i.e., a) compiler and b) interpreter modes.

compartmentalisation, we recognise the following notable examples, each providing various sets of features.

GNU datamash [9] is a command-line tool that performs basic numeric, textual, and statistical operations on textual data. While it is not a fully-fledged DSL, it is suitable for handling basic manipulation and analysis on the input. Compared to CPSLint, it provides general-purpose command-line operations rather than a domain-specific notation for sanitising and compartmentalising industrial time-series data.

Lisp Query Notation [10] is an embedded DSL in Common Lisp that can operate on CSV data as well as structured formats such as JSON. It allows users to query and manipulate textual data both as a DSL and as a command-line tool. Being embedded in Common Lisp enables users to extend its functionality and makes it possible to fall back on Lisp when the DSL is limiting for certain operations. It is more host-language-extensible and in general broader than CPSLint with its small declarative vocabulary tailored to sanitisation, imputation, and compartmentalisation.

DescribeML [11] is a DSL designed to precisely describe machine learning datasets in terms of their structure, provenance, and social concerns, with the aim of providing a method for dataset documentation based on their characteristics, thereby helping data scientists select the appropriate dataset for a given project. DescribeML is implemented using Langium and has been published as a Visual Studio Code extension. Unlike CPSLint, it targets dataset description and documentation rather than executable preprocessing of raw traces.

RADAR [12] is a DSL built for data quality monitoring. RADAR's capabilities include simple checks like making sure there are no null values, data integrity checks and statistical checks like conformance of the data to a certain distribution. This DSL is made up of 3 types of definitions; sources (referring to the input data), checks (which act like functions that accept parameters), and actions (which apply the checks on data from a source and define what should be reported for the resulting values). In contrast to CPSLint, RADAR emphasises monitoring and reporting of data quality rather than transforming corrupted data into sanitised output datasets.

Lavoisier [13] is a DSL that helps with creation of datasets that conform to the format accepted by data mining algorithms. It is meant to reduce script size compared to accomplishing the same result using other ways to prepare data such as SQL and Pandas. Its focus is broader dataset preparation for data mining without the CPS focus. *Papin* [14] is a spin-off DSL of Lavoisier that specialises further and makes it compatible with fishbone diagrams, a notation used by industrial engineers to model cause and effect. For the Papin pipeline the authors introduce a new fishbone diagram kind called data-oriented fishbone diagrams which combines domain data (defined using Lavoisier expressions) with standard fishbone diagrams. These diagrams then are used as input for the Papin specification.

Jet [15] is an embedded DSL written in Scala which is meant to be used for batch processing of large datasets. This DSL can generate code for both Apache Spark and Apache Hadoop, both of which are systems for processing big data. The compiler, relational, and domain-specific optimisations of the implementation provide substantial performance gains compared to the naive handwritten processing code. Compared to CPSLint, Jet is much more focused on high performance.

DSL4DPiFS [16] is a graphical DSL to help with deploying data pipelines in industrial metal forming systems, built with interdisciplinary collaboration in mind. These data-pipelines are used to enable process optimisation, quality management, and predictive maintenance of these systems. For CPSLint, we consider the data-pipeline to be an external factor where CPSLint-produced code gets integrated.

KNIME [17] is a graphical platform for building data processing and analytics workflows from interconnected nodes. It supports a broad range of data integration, transformation, analysis, and visualisation tasks in a general-purpose workflow environment. KNIME offers substantially broader workflow support than CPSLint but also requires a longer learning time.

Table 1 shows the position of CPSLint relative to prior work. The comparison indicates that the related tools and DSLs overlap with CPSLint only partially. Some focus on querying, some on monitoring, some on documentation, optimisation, or deployment, but none combines declarative sanitisation, imputation, and compartmentalisation for industrial CPS time-series data in the same way, and only one covers a form of data imputation.

6. Conclusion and future work

We have elaborated the design and implementation of CPSLint, a custom DSL written in the Rascal language workbench. Our DSL is intended to facilitate data sanitisation and compartmentalisation in industrial CPS use-cases, where workflows process data collections in time-series format. We have provided examples of such workflows, in which data parsing, data cutting and ad hoc data sanitisation stages can be collectively replaced by processing through CPSLint. CPSLint as a tool and in its integrated form with workflows from Section 4, can be considered as both a *proof of concept* and a *proof of integration* (based on the definitions by Odyurt et al. [18]) for the ZORRO project.

Table 1
Comparison of supported functionality amongst existing tools/DSLs and with CPSLint in the last row. The considered markings and their meanings are: ●= supported, ◐= unclear/partially supported, and ○= unsupported.

Tool/DSL	Imputation	Type inference	Filtering	Statistical analysis	Data restructuring
GNU datamash	○	○	○	●	●
Lisp Query Notation	○	○	●	○	●
DescribeML	○	●	○	○	○
RADAR	○	◐	●	●	◐
Lavoisier	○	●	◐	○	●
Jet	○	○	◐	○	●
DSL4DPiFS	○	◐	○	○	◐
KNIME	◐	◐	●	●	●
CPSLint	●	●	●	○	●

In its current form, CPSLint is a self-contained tool with sufficient functionality for our demonstration. It can also be easily extended and adapted to the needs of other use-cases, for instance by defining different data headers, or by implementing extra data imputation methods. Such additions can be considered as future extensions.

Acknowledgments

This publication is part of the project ZORRO[2] with project number KICH1.ST02.21.003 of the research programme Key Enabling Technologies (KIC), which is (partly) financed by the Dutch Research Council (NWO).

Declaration on Generative AI

During the preparation of this work, the authors used ChatGPT-5.X in order to perform grammar and spelling checks. Further, the authors used ChatGPT Codex for setting up the CEURART template. After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] M. Mernik, J. Heering, A. M. Sloane, When and How to Develop Domain-Specific Languages, ACM Computing Surveys (2005). doi:10.1145/1118890.1118892.

[2] U. Odyurt, O. Sayilir, M. Stoelinga, V. Zaytsev, CPSLint, 2026. doi:10.5281/zenodo.17406795.

[3] U. Odyurt, Ömer Sayilir, M. Stoelinga, V. Zaytsev, CPSLint: A Domain-Specific Language Providing Data Validation and Sanitisation for Industrial Cyber-Physical Systems, in: Proceedings of the 19th ACM SIGPLAN International Conference on Software Language Engineering (SLE), ACM, 2026. doi:10.1145/3806383.3815519.

[4] U. Odyurt, J. Roeder, A. D. Pimentel, I. G. Alonso, C. de Laat, Power Passports for Fault Tolerance: Anomaly Detection in Industrial CPS Using Electrical EFB, in: 2021 4th IEEE International Conference on Industrial Cyber-Physical Systems (ICPS), 2021. doi:10.1109/ICPS49255.2021.9468262.

[5] P. Klint, T. van der Storm, J. Vinju, RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation, in: 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, 2009. doi:10.1109/SCAM.2009.28.

[6] P. Klint, T. van der Storm, J. Vinju, EASY Meta-programming with Rascal, in: Generative and Transformational Techniques in Software Engineering III: International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers, Springer, 2011. doi:10.1007/978-3-642-18023-1_6.

[7] U. Odyurt, D. Sapra, A. D. Pimentel, The Choice of AI Matters: Alternative Machine Learning Approaches for CPS Anomalies, in: Advances and Trends in Artificial Intelligence. From Theory to Practice, 2021. doi:10.1007/978-3-030-79463-7_40.

[8] S. Erdweg, T. van der Storm, M. Völter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. P. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, J. van der Woning, Evaluating and comparing language workbenches: Existing results and benchmarks for the future, Computer Languages, Systems and Structures (2015). doi:10.1016/J.CL.2015.08.007.

[9] GNU Project, GNU datamash, 2025. URL: https://www.gnu.org/software/datamash/.

[10] A. Hoff, Lisp Query Notation — A DSL for Data Processing, 2024. doi:10.5281/zenodo.11001584.

[11] J. Giner-Miguelez, A. Gómez, J. Cabot, A Domain-Specific Language for Describing Machine Learning Datasets, Journal of Computer Languages (2023). doi:10.1016/j.cola.2023.101209.

[12] F. Heine, C. Kleiner, T. Oelsner, A DSL for Automated Data Quality Monitoring, in: Database and Expert Systems Applications, 2020. doi:10.1007/978-3-030-59003-1_6.

[13] A. de la Vega, D. García-Saiz, M. Zorrilla, P. Sánchez, Lavoisier: A DSL for Increasing the Level of Abstraction of Data Selection and Formatting in Data Mining, Journal of Computer Languages (2020). doi:10.1016/j.cola.2020.100987.

[14] B. Sal, D. García-Saiz, A. de la Vega, P. Sánchez, Domain-Specific Languages for the Automated Generation of Datasets for Industry 4.0 Applications, Journal of Industrial Information Integration (2024). doi:10.1016/j.jii.2024.100657.

[15] S. Ackermann, V. Jovanovic, T. Rompf, M. Odersky, Jet: An Embedded DSL for High Performance Big Data Processing, 2012. URL: https://infoscience.epfl.ch/handle/20.500.14299/85985.

[16] B. Vogel-Heuser, M. Zhang, M. Krüger, A. Vicaria,

[2] https://zorro-project.nl

M. Gardill, Y. Jiang, A. Trächtler, H. Peters, M. Liewald, A. Schenek, P. Heinzelmann, M. Weyrich, DSL4DPiFS — A Graphical Notation to Model Data Pipeline Deployment in Forming Systems, at - Automatisierungstechnik (2025). doi:10.1515/auto-2024-0114.

[17] M. R. Berthold, N. Cebron, F. Dill, T. R. Gabriel, T. Kötter, T. Meinl, P. Ohl, K. Thiel, B. Wiswedel, KNIME — the Konstanz information miner: version 2.0 and beyond, SIGKDD Explorations Newsletter (2009). doi:10.1145/1656274.1656280.

[18] U. Odyurt, R. Loendersloot, T. Tinga, Demonstrators for Industrial Cyber-Physical System Research: A Requirements Hierarchy Driven by Software-Intensive Design, 2026. doi:10.48550/arXiv.2510.18534.