Domain-Specific Visual Language for Data Engineering Quality

Alexis De Meo
Trove
Wellington, New Zealand

Michael Homer
mwh@ecs.vuw.ac.nz
School of Engineering and Computer Science
Victoria University of Wellington
Wellington, New Zealand

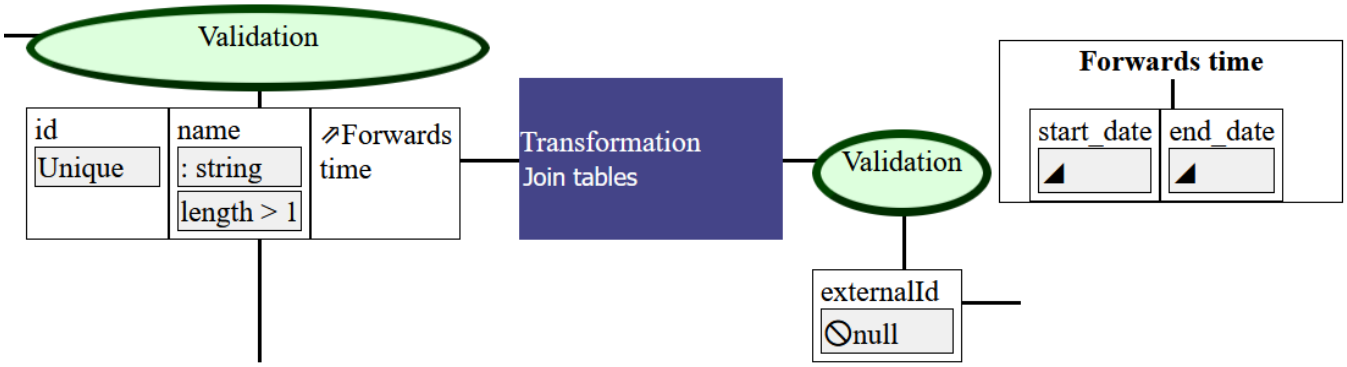

Figure 1. A very simplistic data quality validation pattern diagram in our language.

Abstract

Data engineering pipelines process large amounts of information, and ensuring that the quality and integrity of the data is maintained throughout is critical for technical, business, and social reasons. Conventional data quality assurance approaches require a large amount of fine-grained testing code, which is laborious, easy to get out of sync, and inscrutable to non-technical stakeholders. An executable higher-level visual approach to expressing quality requirements can serve as a shared representation of these constraints and their implications for all parties, eliminating repetition while increasing accessibility and maintainability. We present a visual programming language for expressing data quality requirements within a pipeline declaratively, structured as a diagram of compositional data flow, transformation, and validation steps.

CCS Concepts: • **Software and its engineering → Visual languages**; *Data flow languages.*

Keywords: data engineering, visual programming, dataflow programming

ACM Reference Format:
Alexis De Meo and Michael Homer. 2022. Domain-Specific Visual Language for Data Engineering Quality. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments (PAINT '22), December 05, 2022, Auckland, New Zealand*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3563836.3568727

1 Introduction

Data engineering involves moving and transforming data from its source to destinations via *pipelines* for further analysis and use. There are many phases of this process, including data cleaning, standardisation, obfuscation, transformation, integration, and storage. Data may be ingested from numerous upstream sources, including both other units within the organisation and third parties. A key objective of data engineering is to ensure that the data arrives meeting certain standards and requirements, such as ensuring that all expected information is present, that each piece of information is in the correct format, and that data stored in different locations is consistent, and so on, so that consumers of the data can use it in a meaningful way. Ensuring that these requirements are met is what *data quality* is about.

What data quality means will be unique to its source and consumption, and deciding on this meaning involves multiple parties, including diverse stakeholders elsewhere in

PAINT '22, December 05, 2022, Auckland, New Zealand

the organisation. In practice, the data quality requirements are often expressed in program code, which is not easily accessible to non-technical stakeholders.

There are existing visual languages for data engineering, which focus on allowing non- or less-technical users to construct pipelines and perform analyses in a "no-code" graphical environment. These systems are generally totalising, controlling the full scope of the process, but there is an absence of explicit data-quality validation within them, and by expressing the many "programming" steps of data pipelines they become inaccessible to non-technical parties.

More often, these pipelines are constructed using conventional languages, which are also not accessible for non-technical stakeholders. Data quality can be checked by program code, often using libraries built for this purpose. Within a data pipeline, quality validation can happen at multiple points, and so quality-checking code is interleaved with cleaning, standardisation, and transformation logic.

We make two arguments:

1. Visual data-engineering languages should include explicit quality-checking constructs, as building quality checks within them currently is overly fraught and often omitted.
2. For pipelines built primarily in conventional languages, an executable diagrammatic form structured around quality checks, with the operational logic continuing to be built as before, is more useful than a purely textual representation.

We propose a novel domain-specific visual language for expressing these quality requirements, and the tasks that accompany verifying and enforcing them, drawing on prior work in visual languages, pattern matching, and data engineering. This diagrammatic form is intended to translate to executable code on the appropriate platforms, while being at least understandable to both data engineers and non-technical stakeholders, as well as to software engineers without data engineering expertise. In this system we can express both the sorts of requirements that are important and some of the pipeline structure to deal with the verified data, including directing output data to either storage or further processing steps written in other languages.

The contributions of this paper are:

- The design of a visual pattern-matching language for expressing data quality requirements and the tasks that accompany verifying and enforcing them.
- A prototype frontend for this language.

To do this, we draw on industrial experience both building and — crucially — maintaining data pipelines, and integrate it with existing literature on visual programming and data engineering. The next section discusses key concepts of data engineering and data quality for a new reader, and addresses the state of the art. Section 3 then presents the design of our visual language, while Section 4 describes our prototype frontend and potential future work.

2 Background and Related Work

2.1 Data Engineering

Data Engineering involves moving and transforming data from its sources to destinations via *pipelines* for further analysis and use. Data arriving from upstream may be in a variety of formats, including JSON, column formats, or XML, among other formats, often with heterogeneous structure. The volume and velocity of data can be significant, requiring specialised high-throughput and scalable systems and techniques, and it is common to have a polyglot environment where different points in the pipeline use different languages or platforms to suit task-specific needs. A data source could be another unit of the same organisation, a third-party service procured by the organisation, or data collected without direct collaboration from the source at all. Typical data engineering workflows will involve creating and maintaining innumerable pipelines, each of which involves one or more sources and destinations, and potentially multiple intermediate transformation steps. It may or may not be possible to coordinate with data producers on any changes desired or intended, and in many cases the data consumption is an ancillary outcome from the upstream perspective: for example, data may be collected on accounts being created, in order to analyse the information further, but from the perspective of the implementors of the account-registration system the focus and purpose of their task is accomplished once the account record is committed.

As a result, changes may occur across the upstream system with impactful effects on the consumer, but with all producer-side code made consistent at the same time: a field is renamed, a data type is widened or narrowed, additional information is recorded, or made optional. Alternatively, a source system may be decommissioned entirely, or replaced, such that the pipeline dries up.

2.2 Data Quality

Data Quality covers a wide range of topics, including statistical validity, collection methodology, and integrity retention, and has been defined as "the measure of the agreement between the data views presented by an information system and that same data in the real world" [9, 21]. The main focus in this paper is data quality in the sense of ensuring the integrity of transformations within a data pipeline, and detecting potential issues upstream that may compromise this; it addresses data quality dimensions [28] of completeness, consistency, and validity, among others.

Data quality requirements come from a variety of sources, and measuring them can be complex and manual [1, 16].

Some are purely technical: there is code that expects a certain structure, or a certain type, and will fail if that expectation is not met. Others are more business-oriented: to ensure that analytical processing sees an accurate picture, certain consistency requirements must be met, and the data must be complete and accurate. Still others are regulatory: certain data must not be included in analyses, such as data that identifies individuals (PII) [19], and certain records must be retained for a period of time. All of these requirements must be met, and the data engineering pipeline must be able to detect when they are not, and to correctly handle the situation, preventing bad data from propagating [17]. This validation must happen *at run time*: data pipelines ingest data from a variety of sources, and are generally automated; it is not feasible to rely on static checking or upstream cooperation. It must also happen *in production*: that is where the real data can be found, and where any problems that arise are critical; advanced testing, type systems, and static analysis can be useful, but any such quality checks must be replicated in production anyway.

2.2.1 Why Not Just a Type System?

Data quality requirements encompass more than conventional type systems and database schemata cover, and the required solutions have more range and variation than a direct rejection or correct-by-construction mindset. Simple requirements that a field exists in the input data, that the field values have a specific type, or that values in the field are not null, are analogous to typical type systems and schemata. However, there can be substantially more complex requirements, ranging from specific structures expected to be within formally-unstructured fields (this string must be a valid email address) to multi-field constraints (the sum of values in fields B, C, and D must be equal to the value in column A), or requirements across the entire data batch: the proportion of outlier values in column A must be within 10% of the proportion across all batches of data to date. The necessary actions when a requirement is not met are also varied: log for observability, reject the batch, reject the row, reject all future data, and are necessarily expressed through code to address this variety.

Similarly, constraints in database management systems can express some data quality requirements — but only at the point where the data has already reached the storage stage. This is too late for many requirements, and cannot address intermediate transformations that are never persisted. They also have very limited ability to enforce more complex batch-level requirements, such as defect rates, or to support the various actions that a failed quality requirement may need. However, where a requirement *is* at the point of storage, and where the data is not subject to further transformations, realising the quality requirements as database constraints can be a useful and efficient approach, but this is an implementation detail.

2.3 Visual Data-flow Languages

Data-flow or flow-based programming lends itself to graphical representations [7, 18]. Most often, these follow a "node-and-wire" approach, where individual boxes represent a transformation step, and connections between them indicate the flow of data. There are numerous commercial general-purpose systems in this mold, such as Simulink [27] and Pure Data [2]. Such systems give a direct expression of the processing order and pathways, but programs can rapidly become overly complex and difficult to read [25].

These languages have a range of well-traversed issues [23], including user friction, a lack of well-typedness in composed interactions [6], and difficult interactions with typical software-engineering tools [3]. However, they have also been upheld as supporting the direct expressions of common tasks in a variety of fields [13, 14], and domain-specific nodes have been advocated to ameliorate difficulties for concrete cases [4, 22].

2.3.1 Existing Visual ETL systems.

There are a number of existing visual languages for extract-transform-and-load and data analysis pipelines, such as KNIME, Enso, and Orange. These endeavour to cover the full length of the pipeline, but have received criticism from practitioners [10] for both reasons typical for visual languages (e.g. space, maintainability, wire-crossing, onerous interaction, only looking good in a sales presentation, inflated expectations of non-programmer utility), and those specific to data engineering workflows: data governance concerns, fostering or requiring non-modular repetition in the inherently recurring tasks of pipelines, totalising approaches to integration with other systems, and fostering neglect of unexpected data.

It is possible to express quality checks in most of these systems through combinations of standard conditional branching and filtering, just as it is in textual languages, but made worse by the inefficient interaction and representation of the visual system. It is usual to have many requirements to check at once, and to need all of them to be met in order for the pipeline to continue. This is a difficult problem to express in these languages, and encourages only the most superficial or obvious checks to be made. Quality checks are also inextricably enmeshed with business-logic connections and difficult to identify. What we propose here could be incorporated within existing visual languages as self-contained blocks, explicitly about asserting quality properties of data.

We argue that data quality validation is necessarily distinct and meaningfully separate from other conditional branching and filtering operations, and should be both explicitly marked as quality checks and expressed in a manner tailored for the needs of quality validation, rather than being performed using unmarked general-purpose operations. Explicit data-quality checkpoints are analogous to explicit type annotations in other situations, both for drawing attention to themselves and the expectations they embody when they

are important, and for being easily glossed over when they are not.

2.4 Data Quality Frameworks for Textual Languages

Data quality frameworks such as Great Expectations [20] and Deequ [24] provide a framework for expressing these requirements, but require meticulous expression of them in program code. It is easy for the code to become out of sync with the requirements, or with other code that depends on the same data. They are also not designed to be used by non-technical stakeholders, and are not easily translatable to other languages, so the polyglot environments that are common in this space are not well supported. These frameworks focus on local enforcement of requirements, and can be difficult to apply in the context of a distributed pipeline, operating in a run-time ephemeral environment decoupled from the data storage.

In the next section, we outline the domain-specific visual language we have designed to express these requirements, and to connect to the necessary proprietary systems in other languages that implement the actual behaviour of the data pipelines in these contexts.

2.5 Pattern Matching

The task of validating data quality is very reminiscent of pattern-matching, a common feature in functional programming languages that has more recently been incorporated into object-oriented languages [5, 8, 12, 15]. Pattern matching is fundamentally about taking an unknown (compound) value and inspecting it dynamically to establish its structure and properties, and then branching code execution accordingly. In common pattern matching, multiple patterns may be nested and matching all of them is required to succeed. Data quality checks have some additional needs beyond common pattern-matching tasks. In particular:

- There are usually many different requirements to be applied at once to the same data.
- The values in question are always compound values, with many internal fields, and have no nominal label.
- For any individual field/column, there are likely to be several simultaneous requirements.
- Quality requirements span both individual data items and the entire batch of data, and may both be required at once.
- Bespoke application-specific requirements may be necessary to express the required conditions, not only type, structural, and typical dependent checks.

For our purposes we need a powerful extensible pattern-matching system akin to those of Scala [5], Grace [11], or F# [26], but with a visual representation that is at once expressive enough, sufficiently compact to be tractable, and broadly comprehensible.

3 Data Quality Validation in Visual Form

Our diagrams have three main audiences:

- data engineers, who build the pipelines and must be able to specify and rely on data quality;
- non-technical stakeholders, who must understand that their requirements are accounted for;
- and computers, which must carry out the necessary work to execute the intent of the first two.

All three of these are important, but in particular we argue that *executable diagrams* are crucial: not only must the people understand and manipulate the diagrams, but it is vital that the diagrams not be able to get out of sync with the code that implements them if they are to be relied upon.

Existing visual tools for these pipelines do not incorporate explicit data-quality validation, and most pipelines are still built using conventional languages and tooling. What we propose does not replace any of this: rather, it is a language mechanism for expressing these validation requirements that could either be incorporated within another visual language, or layered on top of conventional tooling. Either way, the transformation steps that "do the work" can remain as-is. However, quality checking, which requires laborious nested conditionals in existing visual systems, or else is delegated to opaque blocks of conventional code, can be expressed concisely with a careful design of a node with that purpose.

These diagrams do not replace the whole pipeline, and do not replace the traditional code with which the pipeline stages are currently built, or existing visual languages that do not incorporate data quality validation in this way. Rather, they are a new layer of abstraction, which can be used to express the data quality requirements and the tasks that accompany verifying and enforcing them, while delegating the body of the work to where it is currently implemented. What has moved is the implementation of the data-quality requirements: rather than being implemented in code as a preamble to each implementation, repeated for each stage consuming the same raw data, they are expressed in the diagrams, and the code is generated from them.

This domain-specific visual language follows a "flow chart" or "nodes and wires" structure, but with complexity concentrated within the nodes. It is necessary to be able to express the gamut of data-quality requirements, and to express a large number of such requirements concisely in one place. It is also necessary to express where data should be directed after validation, and potentially in turn which quality requirements apply to the results of such a transformation step.

The following subsections introduce the major nodes of the language, with illustrations of the kinds of cases they are intended for.

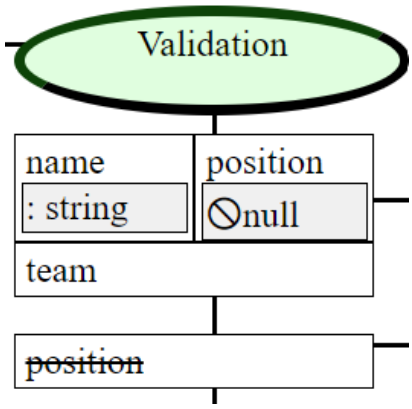

Figure 2. A validation phase and two single classifications below. The first has three specified requirements. The "name" field is required to be a string, the "position" field non-null, but the "team" field only to exist. If all of these criteria are met, data will flow to the right. Otherwise, the next classification below will be assessed, and require that the "position" field not exist.

3.1 Validating and Classifying

An arriving batch of data has no known structure initially, but one or more structures will be expected and acceptable. The "validation" phase attempts to classify incoming data against given sets of criteria. Below the "Validation" node, several classifications can be defined, which are assessed from top to bottom with the first successful classification determining the validation result.

The "classification" nodes are where the most significant part of our DSVL operates. Many different requirements may be part of the same classification, and all must be met for that classification to be successful. Each classification node corresponds to a single "case" in a traditional pattern-matching system, and the combined requirements are akin to recursive destructuring matches. The expectation is that any given classification will specify constraints on several fields of the data, potentially with multiple specific requirements each, but even a single requirement may be useful.

The available types of requirement are discussed in the next section. Figure 2 depicts a simple validation node. Its topmost case has three basic requirements. All of the requirements must be satisfied for the case to match, and when they are the diagram indicates where this batch of data should be directed via the connection to the right. Otherwise, the case connected below is tried, and so on, so the vertical structure represents the "or" pathway, also as in common pattern-matching syntax.

Most often, the topmost case will be the expected structure, and no further inspection will be required in the common situation, and there may even be only a single case to handle. However, it is possible that there are multiple acceptable structures, for example when both a historical and a new format is processed by the same pipeline. In that situation, the different formats may be directed to different places for normalisation, or may be sent directly to the same destination. It may also happen that specific kinds of requirement failures are to be treated differently: for example, an empty value may at times only require logging, and the data can continue through the pipeline regardless, or it may be a *more* severe kind of failure than others for this particular pipeline; either situation can be addressed in this way.

If no case matches, the match fails, and one of the failure modes is engaged.

3.2 Requirements

There are three major families of requirement, each constraining a different aspect of the data. The three deal with

- A single field's existence and properties, such as its type.
- Properties of an entire column of data, e.g. uniqueness.
- Properties of a multiple fields in combination, such as summing to a particular value. This is the most varied type of requirement.

Each of these is discussed below, and alongside, additional affixes can be applied to specify the nature of the constraints (for example, type, within a range, or non-null). We specify these *below* the top-level requirement. Multiple of these may be applied to a single requirement, to specify both a field's type and its valid range, for example.

3.2.1 Field Existence and Properties. The simplest requirement is that a field or column by a particular name must exist in the data. This is liable to be the most common requirement in most cases, and consequently it has the simplest representation: solely the name of the field.

In the simplest case, that is the entire requirement, but often there will be other needs, perhaps most commonly requiring a particular type. We represent this constraint as ": type-name", placed below the field name. Figure 2 shows this value constraint on the "name" field. Another common need is to ensure that the field is not null, seen on the "position" field, or to require a numeric field to be greater or less than a specified value, shown with e.g. "< 100".

Many other such constraints are possible, including highly-bespoke ones fitting within a particular system. For example, a string field might be required to be a valid email address, or to contain an ISBN, a number may be required to be a valid postal code, or a date might be required to be a weekday.

In any case, a field requirement is assessed against each item in the data individually. If *any* of the items do not have the field, or if an additional constraint is not met, then the requirement fails and so will the classification it is a part of. However, data cannot always be perfect, and at times it is useful to indicate a level of failure that is acceptable. Figure 3 shows a field that is required to be at least 75% non-null.

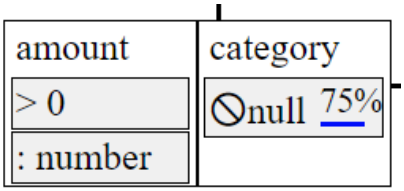

Figure 3. Numerical field required to be greater than a specified value, while another one can be no more than 25% null.

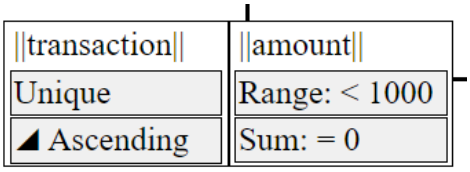

Figure 4. Classification for columns, each with two specific constraints. The column is indicated by name between the paired lines, while the affixes appear in the same fashion as for individual fields.

A field may also be required *not* to exist, represented by the column name alone with a line through it. In this case, there are no further affixes available.

3.2.2 Column Properties. Other requirements may apply to the entire column of data, such as uniqueness, being in ascending order, or having a limited min-max range.

These properties can only be assessed across the entire batch, and not for individual data items. Requirements across the column are indicated by double vertical lines around the name, and the additional constraints are placed below in the same fashion as for individual fields.

Figure 4 shows two column requirements:

- The "transaction" column must contain unique values, and they must be in ascending order.
- The "amount" column must sum to zero, and the minimum and maximum values must be less than 1000 apart.

Most affixed constraints for columns are not applicable to individual fields, and vice-versa, but some may be shared.

A particular form of this requirement is that there must be enough data points in the batch (for example, to ensure statistical significance). In this case, the column itself is not relevant, but any ubiquitous column may be specified.

3.2.3 Multi-field Combinations. Some requirements operate over multiple values within a row of data. For example, there may be multiple percentages that must add to 100%, or one may need to be less than another. These requirements can be the most varied of all, and take two main formats.

In the first, the values of two or more columns may be combined to form a new value, and that value may then be constrained in the same way that an individual field is. In this case, the fields involved and the combining operation form the top-level requirement, and the constraints on the resulting value laid out below. For example, Figure 5 shows that "work", "play", "sleep" must sum to 100%.

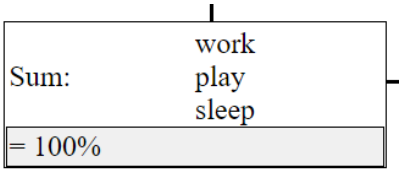

Figure 5. Requiring the sum of three columns to be 100%.

The other format is that the values of two or more fields may be related to one another somehow: the value of one is found in the other, one is less than the other, and so on. Such a requirement has no value itself, but only succeeds or fails based on the input, so there are no affixes available.

3.2.4 Named References. There will often be sets of requirements that are reused in multiple places, to ensure that their qualities are preserved throughout processing. To avoid repetition, it is possible to give a name to a collection of requirements, and then refer to that name in multiple places. These are akin to type declarations, albeit of the very dependent nature that these classifications impose.

Within the diagram, these definitions are placed off to the side, separated from the main flow of the diagram, with a name above. Within a classification, the name given to the collection is used as a single element, and incorporates by reference all of the defined elements. Additional requirements may be given around it as usual.

As well as being more concise, these can ensure that the same requirements truly are applied in multiple places, that they are consistent, and that any modifications reach every relevant place.

3.2.5 Aggregate Values. Columns may sometimes have compound types, like lists, sets, or dictionaries. There can be data-quality requirements on the values within such structures, and on the structure itself.

Requirements on the inner values must be encapsulated within a named definition, and a requirement can then apply that definition across the compound value's contents. This prevents the requirements from becoming arbitrarily nested and incomprehensible, as could happen if they were directly included in situ.

Requirements on the aggregate value itself, such as the number of elements, are expressed via affixes on a standard field requirement. These affixes will overlap with those relevant for all-of-column requirements, but some will only be suitable for one or the other.

3.3 Transformation

Transformation phases represent the places where code runs that manipulates the data, rather than validating it. They may be connected to validation phases on either side, but we do not define exactly how they should work here: perhaps many nodes within a totalising visual system, or only a reference to textual code. The user interface of this is not material to the design at this level, and we envisage concrete implementations making different choices for how to represent and/or generate code for these phases.

3.4 Expressiveness

We do not focus on anything outside of the data-quality elements themselves here, which is out of scope for this work. The structure of the language supports expressing any requirement of a single value, or of a single column in a batch; multiple groups of requirements can then be given different pathways, but at present always disjoint ones (no duplication). Any transformations of the data are currently left outside of this language, incorporated by reference only. These are deliberate choices for maintaining the scope of this work, but we do not rule out extensions in the future.

4 Future work and Prototype

In addition to purely checking for data quality, the same quality requirements may be suitable for other purposes. For example, it may be desirable to divert or halt the flow of data based on some properties expressed in this notation. This section explores possibilities for this, and introduces our front-end prototype and plans for usability experimentation.

4.1 Separation

At times it is necessary to route data items based on some examination of them, rather than to reject or pause processing. The validation steps from Section 3.1 only decide on a yes or no, but do not allow for allowing *some* of the data to continue. For example, it may be that the rows meeting quality requirements should continue, while others are set aside for manual inspection. In other situations, there may be multiple pathways to take based on properties of the data.

A "Separation" phase has the same structure as Validation, but operates on the level of individual data items: if an item satisfies a case's requirements, it is routed to the destination on the right (batched up with other rows that match the same case), while non-satisfying rows are examined by the case below. Figure 6 shows a separation phase with two cases, a standard path for values with a non-null position (expressed in the same way as validating the same property), and a different path for those with a null value there.

All of the same requirements are available, except for those relating to the batch as a whole and to entire columns, and they are used and represented in the same way.

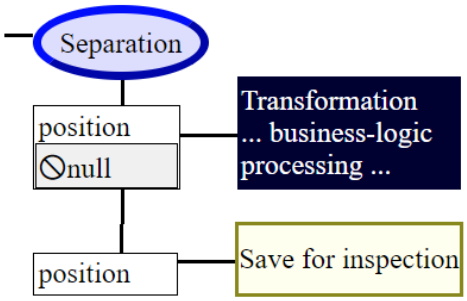

Figure 6. This Separation phase sends data with a non-null position on for further business-logic processing (elided). It sends any other data with a position key to storage for inspection by the data engineer, while values without even that are discarded here.

4.2 Quality Dams

Beyond simply measuring the data quality, we suggest one further feature for systems concerned with preserving the quality of data: the ability for a Validation phase to stop the flow *indefinitely* if a serious quality issue is detected. We call this a "quality dam", and it will ordinarily allow data to flow through uninterrupted. If closed, however, it will enqueue all incoming data and not permit any further processing. These sorts of queues are common in practical data pipelines, for both semantic and performance reasons, but we suggest that explicitly tying them to quality requirements is a good idea.

For example, a Validation phase may detect a serious problem with a batch of incoming data suggesting an unanticipated upstream change, or it may be crucial that the data be processed in a particular order. In this case, permitting future batches of data to flow through the pipeline when they happen not to fail the same quality requirement would lead to unreliable analysis or other negative outcomes.

A Validation phase or one of its classifications should be able to trigger the closure of this dam explicitly in order that a data engineer can intervene before further automated processing. Implementing this sort of backlog on an ad-hoc basis is unreliable, but quality validation is the precise situation that ought to trigger it, and these — or something similar — should be part of any structured data-quality system.

4.3 Data Cleaning and Standardisation

While this paper has focused on the later stages of data pipelines, these tools could also be useful in the data cleaning and standardisation stages. Inspecting incoming data and performing cleaning and standardisation operations, such as removing PII or standardising values, could be aided with the Validation or Separation phases to detect when these operations are needed.

Some data cleaning and standardisation operations align well with the requirements expressed in this system, such as null value handling, simple column renamings or deletions, or additional columns derived from multi-field combinations, or normalisation of values. For both space and focus we limit this work to after the initial cleaning phase, but simple reshaping operations employing elements of this notation are a possible extension.

Indeed, in the case of some quality requirements there is an obvious step to make them true, such as case normalisation, or removing a column that should not exist, while in others a small addition could specify a default value to use. However, some cleaning operations can bleed more into the area of the pipeline's business logic, and are often not relevant to some of the stakeholders, so we have some caution about incorporating them.

4.4 Prototype

We have constructed a front-end prototype of a diagram editor for this language, available on the web at http://ecs.vuw.ac.nz/~mwh/demos/dsvl-deq/ and used to create our example figures. This prototype is able to simulate the evaluation of a pipeline with these validation and separation phases, and a small number of additional processing steps for ease of testing. Data can be imported as a JSON file containing an array of objects, and the pipeline can be exported to a separate JSON file for consumption by other tools interfacing with the proprietary backends.

4.5 User Research

The key next step for this work is to conduct user studies with appropriate audiences to measure which elements of the language are useful, or not. We have not yet begun this process and present this work as a preliminary step towards a more complete system.

5 Conclusion

Data engineering pipelines must preserve the quality of data throughout, and detect introduced flaws in data as early as possible. Expressing these quality requirements in existing systems is either awkwardly complex, or inscrutable to the non-technical stakeholders who participate in determining the needs of the application — or both. A tailored visual language for expressing these quality constraints in executable diagrams can ensure that they are easily understood, always up-to-date, and more readily expressed than existing systems. We presented the design of a system for pattern-matching quality requirements over data batches in a principled fashion, along with some potential extension applications of this approach, and argue that quality requirements are an appropriate place for comprehensible boundaries within data pipelines.

References

[1] Ziawasch Abedjan, Xu Chu, Dong Deng, Raul Castro Fernandez, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, Michael Stonebraker, and Nan Tang. 2016. Detecting Data Errors: Where Are We and What Needs to Be Done? *Proc. VLDB Endow.* 9, 12 (aug 2016), 993–1004. https://doi.org/10.14778/2994509.2994518

[2] Bryan W. C. Chung. 2013. *Multimedia Programming with Pure Data.* Packt Publishing.

[3] Philip T. Cox and Anh Dang. 2010. Semantic Comparison of Structured Visual Dataflow Programs. In *Proceedings of the 3rd International Symposium on Visual Information Communication* (Beijing, China) *(VINCI '10).* Association for Computing Machinery, New York, NY, USA, Article 11, 9 pages. https://doi.org/10.1145/1865841.1865856

[4] Philip T. Cox and Simon Gauvin. 2011. Controlled Dataflow Visual Programming Languages. In *Proceedings of the 2011 Visual Information Communication - International Symposium* (Hong Kong, China) *(VINCI '11).* Association for Computing Machinery, New York, NY, USA, Article 9, 10 pages. https://doi.org/10.1145/2016656.2016665

[5] Burak Emir, Martin Odersky, and John Williams. 2007. Matching Objects with Patterns. In *Proceedings of the 21st European Conference on Object-Oriented Programming* (Berlin, Germany) *(ECOOP'07).* Springer-Verlag, Berlin, Heidelberg, 273–298. http://dl.acm.org/citation.cfm?id=2394758.2394779

[6] Riley Evans, Samantha Frohlich, and Meng Wang. 2022. CircuitFlow: A Domain Specific Language for Dataflow Programming. In *Practical Aspects of Declarative Languages: 24th International Symposium, PADL 2022, Philadelphia, PA, USA, January 17–18, 2022, Proceedings* (Philadelphia, PA, USA). Springer-Verlag, Berlin, Heidelberg, 79–98. https://doi.org/10.1007/978-3-030-94479-7_6

[7] Alex Fukunaga, Wolfgang Pree, and Takayuki Dan Kimura. 1993. Functions as Objects in a Data Flow Based Visual Language. In *Proceedings of the 1993 ACM Conference on Computer Science* (Indianapolis, Indiana, USA) *(CSC '93).* Association for Computing Machinery, New York, NY, USA, 215–220. https://doi.org/10.1145/170791.170832

[8] Felix Geller, Robert Hirschfeld, and Gilad Bracha. 2010. *Pattern Matching for an Object-Oriented and Dynamically Typed Programming Language.* Technical Report 36. Hasso-Plattner-Instituts für Sofwaresystemtechnik an der Universität Potsdam.

[9] Bernd Heinrich, Diana Hristova, Mathias Klier, Alexander Schiller, and Michael Szubartowicz. 2018. Requirements for Data Quality Metrics. *J. Data and Information Quality* 9, 2, Article 12 (jan 2018), 32 pages. https://doi.org/10.1145/3148238

[10] Ian Hellström. 2016. The problems with visual programming languages in data engineering. https://databaseline.tech/the-problems-with-visual-programming-languages-in-data-engineering/.

[11] Michael Homer, Timothy Jones, and James Noble. 2015. From APIs to Languages: Generalising Method Names. In *Dynamic Language Symposium.* https://doi.org/10.1145/2816707.2816708

[12] Michael Homer, James Noble, Kim B. Bruce, Andrew P. Black, and David J. Pearce. 2012. Patterns As Objects in Grace. In *Proceedings of the 8th Symposium on Dynamic Languages* (Tucson, Arizona, USA) *(DLS '12).* ACM, New York, NY, USA, 17–28. https://doi.org/10.1145/2384577.2384581

[13] C. T. Johnston, D. G. Bailey, and P. Lyons. 2006. Towards a Visual Notation for Pipelining in a Visual Programming Language for Programming FPGAs. In *Proceedings of the 7th ACM SIGCHI New Zealand Chapter's International Conference on Computer-Human Interaction: Design Centered HCI* (Christchurch, New Zealand) *(CHINZ '06).* Association for Computing Machinery, New York, NY, USA, 1–9. https://doi.org/10.1145/1152760.1152761

[14] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. 2004. Advances in Dataflow Programming Languages. *ACM Comput. Surv.* 36, 1 (mar 2004), 1–34. https://doi.org/10.1145/1013208.1013209

[15] Tobias Kohn, Guido van Rossum, Gary Brandt Bucher II, Talin, and Ivan Levkivskyi. 2020. Dynamic Pattern Matching with Python. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages* (Virtual, USA) *(DLS 2020)*. Association for Computing Machinery, New York, NY, USA, 85–98. https://doi.org/10.1145/3426422.3426983

[16] Shixia Liu, Gennady Andrienko, Yingcai Wu, Nan Cao, Liu Jiang, Conglei Shi, Yu-Shuen Wang, and Seokhee Hong. 2018. Steering data quality with visual analytics: The complexity challenge. *Visual Informatics* 2, 4 (2018), 191–197. https://doi.org/10.1016/j.visinf.2018.12.001

[17] Nigel Martin, Alexandra Poulovassilis, and Jianing Wang. 2014. A Methodology and Architecture Embedding Quality Assessment in Data Integration. *J. Data and Information Quality* 4, 4, Article 17 (may 2014), 40 pages. https://doi.org/10.1145/2567663

[18] Brad A Myers. 1990. Taxonomies of visual programming and program visualization. *Journal of Visual Languages & Computing* 1, 1 (1990), 97–123.

[19] Arvind Narayanan and Vitaly Shmatikov. 2010. Myths and Fallacies of "Personally Identifiable Information". *Commun. ACM* 53, 6 (jun 2010), 24–26. https://doi.org/10.1145/1743546.1743558

[20] Netlify. 2022. Great Expectations Home Page. https://greatexpectations.io/.

[21] Ken Orr. 1998. Data Quality and Systems Theory. *Commun. ACM* 41, 2 (feb 1998), 66–71. https://doi.org/10.1145/269012.269023

[22] Marco Porta. 2000. Iteration constructs in data-flow visual programming languages. *Computer Languages* 26 (2000), 67–104.

[23] Robert Schaefer. 2011. On the Limits of Visual Programming Languages. *SIGSOFT Softw. Eng. Notes* 36, 2 (mar 2011), 7–8. https://doi.org/10.1145/1943371.1943373

[24] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Biessmann, and Andreas Grafberger. 2018. Automating Large-Scale Data Quality Verification. *Proc. VLDB Endow.* 11, 12 (aug 2018), 1781–1794. https://doi.org/10.14778/3229863.3229867

[25] Marc Schmidt. 2021. Patterns for Visual Programming: With a Focus on Flow-Based Programming Inspired Systems. In *26th European Conference on Pattern Languages of Programs* (Graz, Austria) *(EuroPLoP'21)*. Association for Computing Machinery, New York, NY, USA, Article 6, 7 pages. https://doi.org/10.1145/3489449.3489977

[26] Don Syme, Gregory Neverov, and James Margetson. 2007. Extensible Pattern Matching Via a Lightweight Language Extension. In *ICFP*.

[27] The MathWorks, Inc. 2022. Simulink. https://www.mathworks.com/products/simulink.html.

[28] R.Y. Wang, V.C. Storey, and C.P. Firth. 1995. A framework for analysis of data quality research. *IEEE Transactions on Knowledge and Data Engineering* 7, 4 (1995), 623–640. https://doi.org/10.1109/69.404034

Received 2022-09-01; accepted 2022-10-02