FAIR Data Versioning in Microdata.no

Ørnulf Risnes[1], Bjørn Roar Joneid[2], Eirik Alvær[1], Kenneth Schnelle[2], Ivar Refsdal[1], Vassilios Kalantzakos[2], Kjetil Thuen[1], Sigbjørn Revheim[1], Archana Bidargaddi[1], Johan Sjøberg[2], Marianne Aamodt[2], Trond Pedersen[1], Svein Johansen[2], Pawel Buczek[2], Karin Arar[1], Eirik Stavestrand[1], Rune Gløersen[2]

[1] NSD - Norwegian Centre for Research Data (NSD)
[2] Statistics Norway (SSB)

Abstract

Microdata.no is a research platform for instant access to register data residing in Microdata.no-DataStores.

Supported by integrated and detailed metadata, users interactively develop statistical programs in the form of scripts that are executed immediately on the platform.

Microdata.no-scripts are complete and reproducible text-based records of the analytical results they produce, and therefore easily shareable and citable entities. Repeated executions of a script produce identical results.

In this paper we show how Microdata.no's version control regime for DataStores is designed to support the following ambitions:

1. All data versions are unambiguously identifiable and retrievable from scripts
2. Microdata.no-scripts are citable and reproducible FAIR Digital Objects
3. Users are informed about changes made across all data versions

What is Microdata.no?

Microdata.no is a service and platform designed to simplify access to and increase use of register data in research. It is developed and operated by NSD - Norwegian Centre for Research Data (NSD) and Statistics Norway (SSB).

Norway has many good sources of register data, and Norwegian legislation allows such data to be used for research purposes. However, the traditional application process for obtaining de-identified microdata from registers is complicated, and adapting datasets for researchers is time-consuming and costly.

Microdata.no gives researchers instant and flexible access to detailed, full population register data and enables them to use the data without going through the application process.

The European Commission has guidelines stating that research data should be ‘as open as possible, as closed as necessary’. In Microdata.no, register data can be used immediately and flexibly without compromising privacy, demonstrating how traditional access has made such data *more* closed than necessary, and how to open them up safely and securely.

The core FAIR Data Principles[1], drafted in 2015, have received worldwide recognition and have been promoted by the European Commission as a framework for data sharing and work procedures with maximum transparency, use and reuse. Microdata.no is designed, developed and deployed to support existing and emerging policies surrounding FAIR Data Principles.

How does Microdata.no work?

Accredited researchers use a web-based client ROSE (RAIRD Online Statistical Environment) to interact with data and metadata available on the platform.

ROSE is the user-facing part of Microdata.no’s data-anonymizing interface. ROSE-users never see data directly, but can find, select, operate on, transform and analyze data through the client.

ROSE is script-based. Users build up scripts interactively in ROSE’s command-line or batch-oriented in the script-editor, or by a combination of the two.

DataStores, data, metadata and client integration

To fully understand how data versioning works in Microdata.no, an understanding of the platform’s tight integration between data and metadata is needed.

Microdata.no is a metadata-driven system where metadata drives the user interface, and describes the data on the system in detail. Access to data depends on metadata. Metadata is validated against data and vice versa.

The ROSE client is a feature-rich integrated development environment (IDE) that provides an information system to users as well as features such as DataStore browsing, version history browsing, variable metadata browsing, command validation and autocomplete/suggest-features. Most IDE functionality is driven by metadata.

The fundamental abstraction that ties client-behaviour, data and metadata together is called a **DataStore**. In Microdata.no-terms, a DataStore is an autonomously maintained logical collection

[1] **F**indable, **A**ccessible, **I**nteroperable and **R**eusable data

of variables with technical read-only interfaces for version history, metadata and data retrieval and management interfaces for management and data/metadata curation.

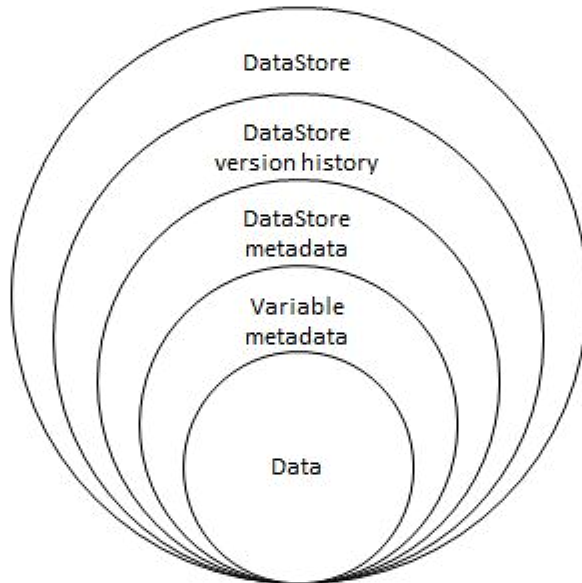

Figure 1 - Logical structure of a DataStore

Figure 1 above shows the logical and conceptual structure of the DataStore abstraction with version history built on top of tightly integrated metadata and data. In practice, a DataStore is the point of access for *all* published versions of its data and metadata as illustrated in Figure 2 below.

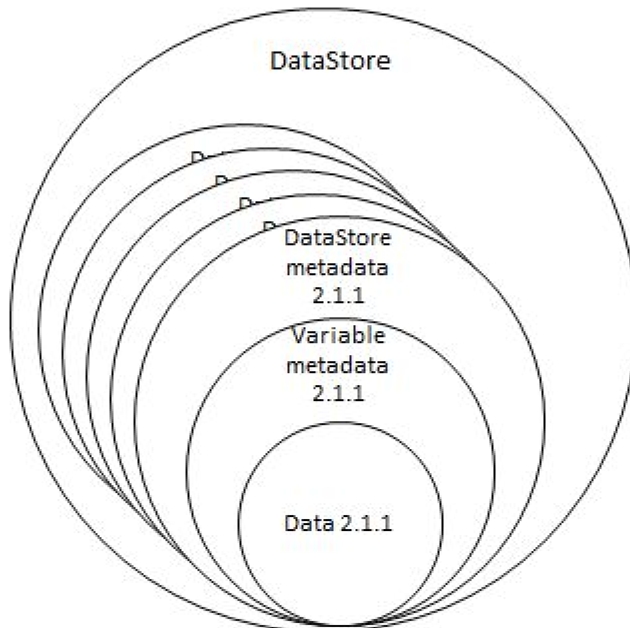

Figure 2 - Sample DataStore with 5 available versions of data/metadata

Self-service

The first time a user logs into ROSE, the user's workspace is empty. ROSE has access to a catalog of read-only DataStores available on the platform.

Users mount (i.e. connect to) DataStores using the `require`-command illustrated in lines 1-3 in Figure 3 below. A script can mount multiple DataStores and multiple versions of the same DataStore.

Users then build up their own custom datasets by importing variables from mounted DataStores into their own private, virtual working datasets. Imported variables that can be merged automatically are merged by the system.

Users can rearrange, sample and transform virtual datasets, as well as create derived variables. Virtual datasets may be analyzed with analytical commands supported by the ROSE scripting language.

Analytical output is subject to automatic statistical disclosure control techniques described in APPENDIX C in Microdata.no's User Manual https://microdata.no/brukermanual-en.pdf.

A script-based approach supports sharing and reproducibility

All data imports, transformations and analyses a given user executes on the Microdata.no-platform gets expressed as scripts, i.e. a sequence of commands expressed in the ROSE scripting language.

```
1  require no.ssb.fdb:3 as fdb3
2  require no.ssb.fdb:2 as fdb2
3  require no.nsd.data:1 as nsd1
4
5  create-dataset vtest
6  import fdb3/SIVSTANDFDT_SIVSTAND 2010-10-10 as sivstand10
7  import fdb2/SIVSTANDFDT_SIVSTAND 2010-10-10 as sivstand10_gml
8  import fdb3/INNTEKT_WSAMINNT 2010-01-01 as saminnt10
9  import fdb2/INNTEKT_WSAMINNT 2010-01-01 as saminnt10_gml
10 import fdb3/BEFOLKNING_KJOENN as kjønn
11 import nsd1/Utdannelse 2010-01-01 as utd_nsd
12
13 summarize saminnt10 saminnt10_gml
14 tabulate sivstand10 sivstand10_gml
```

Figure 3 - Sample Microdata.no-script retrieving data from different DataStore versions

Microdata.no-scripts are solely text-based. Therefore, users may share scripts with colleagues or peers simply by e.g. emailing them the text body of a script. The receiver can, if accredited to use Microdata.no, run the received script and get not only the exact same results as the author did, but the entire stepwise data import and transformation process leading up to the results.

Informally, we can say that a given analytical result is a function of the available data and the script used to produce the analytical result:

$$result = f(data, script)$$

However, as the `require`- and `import`-statements shown in Figure 3 demonstrate, retrieval and usage of specific versions of the version controlled data are *explicitly declared* as part of the scripts.

This enables us to simplify the function and state that a given result is a function of the script alone :

$$result = f(script)$$

The fact that scripts themselves are complete, transparent and easily shareable enables them to be turned into FAIR Digital Objects and shareable entities following the intentions in the European Commision report Turning FAIR Into Reality (https://doi.org/10.2777/54599).

The ongoing Microdata2.0-project will finalize the work making Microdata.no-scripts citable entities further improving Findability, Accessibility, Interoperability and Reusability.

For this to succeed, models, workflows and implementations for data versioning are essential.

Types of changes to a DataStore

Over time, data or metadata held in DataStores can change in ways that can impact analyses *or* their interpretation, or enable new analyses, e.g.:

- addition of new variables
- updates to existing variables with more recent data
- changes to existing variables to correct errors
- changes to metadata

The Microdata.no platform is designed to support reproducible research to the maximum extent possible. Any given script should therefore produce the exact same results over time, regardless of changes to data or metadata.

It follows that *old and new versions of data/metadata must co-exist*, and all versions be reachable from the ROSE environment where scripts are executed.

A typology of changes has been developed. Not all changes impact analyses, results or interpretations. Types of changes fall roughly into two categories:

- *Destructive* changes impact results or their interpretation
- *Non-destructive* changes do not impact results or their interpretation

Destructive changes

Destructive changes can or will influence results or their interpretation.

Without support for versioned data, the following is true:

A script executed after a destructive change *may produce different results* than the exact same script executed before the destructive change.

Destructive changes include:

- Modification of data values in a variable
- Addition of new data records in a variable
- Removal of data records in a variable
- Removal of variables
- Changes to variable metadata that will impact interpretation (e.g. substantial changes to definitions, swapping of value-labels)

Note that, perhaps surprisingly, adding new data records to a variable is classified as a destructive change. This is related to how longitudinal event-data is modeled and represented in the RAIRD Information Model[2], where end-dates for event periods have to be added retrospectively to records that have a succeeding event in the recent data. E.g. a new change in marital status (e.g. Divorced) for a person triggers a change to the end-date of the previous status (e.g. Married).

In the versioning scheme described below, destructive changes are classified as **MAJOR**.

Non-destructive changes

Non-destructive changes will not influence results or their interpretation.

Without support for versioned data, the following is true:

---

[2] https://statswiki.unece.org/display/gsim/RAIRD+Information+Model+RIM+v1_0

A script executed after an non-destructive change *will produce identical results* to the exact same script executed before the non-destructive change.

Support for non-destructive changes is there to support users with information and change logs of all changes to data and metadata.

Non-destructive changes include:

- Addition of new variables
- Expansion of variable metadata
- Unsubstantial changes to metadata (correcting typing errors, etc)

In the versioning scheme described below addition of new variables and metadata expansion are classified as **MINOR** whereas unsubstantial metadata changes are classified as **PATCH**.

Versioning scheme

Microdata.no DataStores use a versioning scheme inspired by semantic versioning[3] (SemVer), where a DataStore version number is made up of **MAJOR**.**MINOR**.**PATCH**, where

- **MAJOR** is incremented for destructive changes
- **MINOR** is incremented for non-destructive changes users should know about
- **PATCH** is incremented for non-destructive changes of little relevance to users

See Table 1 below for examples.

Version number	Change description
1.0.0	Initial release of the DataStore
2.0.0	Data records for new time periods added to existing variables.
2.1.0	New variables added. No changes to existing variables.
2.2.0	More new variables added. No changes to existing variables.
2.2.1	Correction of metadata typos

Table 1 - Examples of version changes to a Microdata.no DataStore

[3] https://semver.org/

A SemVer-based versioning scheme was evaluated against a more Git-like approach (version numbers are hashes of contents) and a timestamp-based approach.

SemVer is not without weaknesses, but was chosen mainly because SemVer version numbers have an explicit sequentiality (unlike Git hashes) and a format that supports explicit differentiation between destructive and non-destructive changes (unlike timestamp-based versioning schemes).

Automated version number updates

A DataStore is maintained by an individual or a group. Over time, maintainers will change the contents of the DataStore, likely in a collaborative manner.

Manual updates of the DataStore's SemVer version number will be error-prone and time-consuming.

An automated regime is implemented to support automatic version number incrementation upon releasing a new DataStore version into the Microdata.no platform.

Every change is an effect of a change event performed by a maintainer. All possible change events are operationalized and modeled.

Operationalizing change events

Table 2 below shows the different events DataStore maintainers can perform and their effects on the version number incrementation.

Change event	Description	Increments
ADD	A variable (data and metadata) has been added to the DataStore	MINOR
REMOVE	A variable (data and metadata) has been removed from the DataStore	MAJOR
CHANGE_DATA	Data values has been changed or removed	MAJOR
ADD_METADATA	More metadata has been added	MINOR
CHANGE_METADATA	Metadata has been changed significantly	MAJOR
PATCH_METADATA	Insignificant change to metadata	PATCH

Table 2 - Types of actions that change DataStores and their effect on the version number

Recording change events and releasing a new DataStore version

When DataStore maintainers modify the contents of their DataStore, all change events described in Table 2 are recorded by the DataStore management tool[4].

A DataStore change event record has 5 columns as shown in the example in Table 3 below.

Maintainer	Variable	Time-stamp	Event type	Change description (optional)
A	VARIABLE_W	T1	ADD	*Income variable added*
B	VARIABLE_X	T2	REMOVE	*Legacy variable removed*
A	VARIABLE_Y	T3	PATCH_METADATA	*Corrected typos in variable definition. Improved English translation of variable info.*
C	VARIABLE_Z	T4	CHANGE_DATA	*Updated with 2019 data.*

Table 3 - Example of a change event record for a DataStore

Automating version number updates

A function to calculate the correct new version number V_{New} for a new release of a DataStore is implemented. The function takes as input the version number of the previous release V_{Prev} and the change event record CER since the previous release.

$$V_{New} = f(V_{Prev}, CER)$$

The logic of the function can be expressed like this (pseudocode):

```
If CER contains 1 or more of ( REMOVE | CHANGE_DATA | CHANGE_METADATA )
=> MAJOR is incremented by 1. MINOR and PATCH are both set to 0.

Else if CER contains 1 or more of ( ADD | ADD_METADATA )
=> MINOR is incremented by 1. PATCH is set to 0.

Else
=> PATCH is incremented by 1.
```

Figure 4 - Pseudocode of version number update function

[4] Working title of the DataStore management tool “Microdata Manager”

Note that the complete CER with descriptions of all changes is available to the system and to end-users. This means that e.g. for a new MAJOR release, information about changes that would themselves trigger MINOR and PATCH increments are described in the CER.

Building an information system on top of versioning

Microdata.no's FAIR Data Versioning is designed to support the following ambitions:

1. All data versions are unambiguously identifiable and retrievable from scripts
2. Microdata.no-scripts are citable and reproducible FAIR Digital Objects
3. Users are informed about changes made across all data versions

Ambitions 1 and 2 are connected, and 2 depends on 1.

Ambition 1 is supported by the combined designs of the DataStore and the ROSE IDE where DataStore versions are mounted explicitly using the `require`-command.

Work to fulfill ambition 2 has been started as part of the ongoing Microdata 2.0-project, and will likely result in an initial release autumn 2020.

Microdata.no-users will then get the opportunity to archive their scripts in a research data repository such as NSD's, associate metadata (author, context, funders, etc), and assign DataCite DOIs to their scripts, pointing to a DOI landing page[5] where their scripts are presented, and their research made Findable, Accessible and Reusable.

Regarding ambition 3;

Users are informed about changes made across all data versions

It is vital to enable users to stay informed about changes as well as to reason about differences between versions.

The data versioning design of Microdata.no supports several mechanisms that can serve these purposes.

An important feature of the SemVer-based versioning scheme and its implementation in Microdata.no is that users only have to refer to MAJOR version numbers when mounting a DataStore. The `require`-commands shown in Figure 3 demonstrate this:

```
require no.ssb.fdb:3 as fdb3
require no.ssb.fdb:2 as fdb2
```

[5] https://support.datacite.org/docs/landing-pages

```
require no.nsd.data:1 as nsd1
```

In the above example, we see that MAJOR versions 3 and 2, respectively, of a specific SSB-DataStore are mounted, followed by a mounting of MAJOR version 1 of an NSD-DataStore.

Since *results of existing scripts by definition cannot be altered by changes classified as MINOR or PATCH*, including more than the MAJOR version number in the mounting statement is unnecessary.

Notifications of new MAJOR and MINOR versions

The versioning scheme is SemVer-based, and the nature of changes between versions can therefore be inferred by looking at the version numbers. This feature makes it trivial to build notification systems on top of the DataStore abstraction, from which version numbers for all published versions are made available.

At this point in time, the plan is to inform users (via e.g. information panels in ROSE) about MAJOR and MINOR changes to DataStores they are actively working on.

When mounting a DataStore for the first time, a user will always be encouraged to mount its latest version. However, it will be the user's decision alone to select their preferred version(s).

Availability of change records

As mentioned above, change event records (CERs) are used by the platform to automatically infer version number increments. But CERs are also of great value to end-users when reasoning about DataStore changes and differences over time. CERs corresponding to the example in Table 3 above are made available in ROSE and other relevant user-interfaces - and via platform DataStore APIs.

Support for quality assurance and to analyse differences and changes between versions

With use and user feedback, data quality typically improves over time. It is likely that newer versions of DataStores will contain data of higher quality than earlier versions. Incomplete or incorrect data may have influenced previous research and analysis carried out on the Microdata.no platform. Enabling users to work with older and newer versions of data in the same setting, will provide tools for them to reason about the effects of previous use of data with reduced quality.

Technology choices supporting data versioning

Many software developers are familiar with the Git[6] source control management system developed by Linus Torvalds.

The data versioning design in Microdata.no supports the notion of immutability throughout the architecture and is highly inspired by key features of Git, including:

- Every version is identifiable and retrievable
- All versions are equally retrievable

The last point is important. Many systems that supports version control treats the current/newest version differently from older versions. Older versions typically need to be reconstructed or in other ways brought from an archived state to an active state.

Git is not like that; in Git all versions are equally retrievable, and there are no performance penalties associated with retrieving an older version of a Git repository.

In Microdata.no, where the user-base likely will work on many different versions at a time, it is crucial that working with old versions performs equally well as working with newer.

However, Git is designed with a file-based source control management perspective in mind, and cannot support Microdata.no's data-metadata integration and behaviour.

Integrated, but separated data and metadata

Early on, the RAIRD-project made a decision to separate storage of metadata from storage of data, while creating tools and mechanisms to keep them synchronized and validated.

There were several reasons for this design choice, including:

- Metadata must be readable from web browsers, data must never be readable from web browsers
- For large datasets, set operations and validations/transformations require specialized software (typically data science tools or relational database engines)

The RAIRD-project's decision to separate metadata from data was however accompanied by another important decision; the only way to reach data is via metadata. This is implemented as illustrated in Figure 1 - Logical structure of a DataStore. Metadata contains the necessary "pointers" required to retrieve data.

[6] https://en.wikipedia.org/wiki/Git

Both DataStore metadata and variable metadata are important in the versioning regime.

Using the Datomic database to hold DataStore metadata and variable metadata

Datomic[7] is a proprietary relational database system with support for immutable data. In short, it is an RDF[8]-inspired database with one universal relation and a sixth normal form[9] with Git-like support for fast retrieval/querying of older versions of the database as well as retrieval/querying of the history of changes to the database.

The latter (**history** queries) is important when creating Change Event Records (CERs); the former (**asOf** queries) is important when bringing up older versions of the database.

In Microdata.no, Datomic is used to hold metadata about the DataStores currently on the platform, and about the variables within them.

History queries provide an information system for software to navigate changes and build CERs and other information objects. AsOf-queries enables querying the metadata database for specific points in time - i.e. the points in time when MAJOR, MINOR or PATCH releases were made.

It is likely possible to create a similar system for version controlled metadata using other technologies. Datomic was selected by the RAIRD project mainly because of its built-in support for version control and its flexible and powerful, RDF-based data model.

It should be noted that it is not trivial to build databases or systems that support the kind of Git-like version control that was identified as a Microdata.no-requirement. Important features to look for when evaluating technologies for these purposes are described in this chapter.

It is beyond the scope of this paper to go further into detail regarding Microdata.no's metadata models, how data are stored and curated, and details about our usage of Datomic.

Future work

The impact of Statistical Disclosure Control on reproducibility of results

Earlier in this paper it was stated that analytical results are but functions of the script that produced them:

[7] https://www.datomic.com/on-prem.html
[8] https://en.wikipedia.org/wiki/Resource_Description_Framework
[9] https://en.wikipedia.org/wiki/Sixth_normal_form

$$result = f(script)$$

In reality, there is at least one other parameter to that function; the configuration of the automated Statistical Disclosure Control (SDC) mechanisms:

$$result = f(script, SDC\text{-}configuration)$$

The first generation of Microdata.no coming out of the RAIRD-project, has one common SDC-configuration for all users and all data.

This is likely to be improved in the ongoing Microdata 2.0-project, where:

- More and improved SDC-techniques will be deployed
- Mechanisms to differentiate the level of SDC on a per-user-basis will be designed, developed and deployed

These will be important developments, and are likely to open up the platform to more data owners and new groups of end users. However, a differentiated SDC-regime will be at odds with the reproducibility ambition laid out in this paper.

Future work will therefore include design of models and systems that can provide auxiliary information that can support other systems and users in reasoning about changes to results that come as a side effect of changed or differentiated SDC configurations.

Such models and systems will likely include change event records also for the SDC-configuration, as well as models that describe SDC-configuration for different users and/or DataStores or some combination of the two.

The impact of completely discontinued/removed DataStores

DataStores are autonomously maintained logical collection of variables. Autonomy represents a risk of discontinuation and/or removal from the Microdata.no platform.

Recommendation 19, action 19.3 in Turning FAIR Into Reality states that

When data are to be deleted as part of selection and prioritisation efforts, metadata about the data and about the deletion decision should be kept. If data deletion is carried out routinely, the underlying protocols for selection and prioritisation need to be made FAIR.

Since DataStore is the abstraction that holds metadata about itself and its variables, mechanisms to retain metadata upon DataStore deletion, and to preserve information about the deletion decision need to be designed and implemented for Microdata.no to support this recommendation.