RegDNA Specification

Contents

[Introduction 3](#_Toc160265864)

[Overview 3](#_Toc160265865)

[Standards based approach 3](#_Toc160265866)

[RegDNA as a simple language 4](#_Toc160265867)

[Tools 4](#_Toc160265868)

[Platforms 4](#_Toc160265869)

[Ecosystems of platforms, tools, and content 5](#_Toc160265870)

[Software-friendly standards 5](#_Toc160265871)

[Auto-generation of software 5](#_Toc160265872)

[Pluggable workbenches 7](#_Toc160265873)

[Don’t re-invent the wheel 7](#_Toc160265874)

[Artefact descriptions 8](#_Toc160265875)

[Model of computation, lineage 8](#_Toc160265876)

[Data Models 8](#_Toc160265877)

[The key roles of models in modern software 8](#_Toc160265878)

[Different modelling approaches 9](#_Toc160265879)

[Software focused models vs database focused models 10](#_Toc160265880)

[Open modelling standards vs closed modelling standards 11](#_Toc160265881)

[Reuse of open standards 11](#_Toc160265882)

[Terminology 11](#_Toc160265883)

[Input Layer 12](#_Toc160265884)

[Enriched Input Layer 12](#_Toc160265885)

[Output Layers 12](#_Toc160265886)

[Datamodel examples in RegDNA 13](#_Toc160265887)

[Version differences 15](#_Toc160265888)

[Logical datamodels and disjoint subtyping 15](#_Toc160265889)

[Logical Transformations 18](#_Toc160265890)

[Introduction 18](#_Toc160265891)

[Derivation transformations 19](#_Toc160265892)

[Derivations that are single attribute enrichments 19](#_Toc160265893)

[overview 19](#_Toc160265894)

[example 19](#_Toc160265895)

[dependencies 19](#_Toc160265896)

[Platform model of computation 19](#_Toc160265897)

[Derivations that are not single attribute enrichment 20](#_Toc160265898)

[Examples from Corep 20](#_Toc160265899)

[Representation in RegDNA 21](#_Toc160265900)

[Example 21](#_Toc160265901)

[Example of Lineage 22](#_Toc160265902)

[Generation transformations 23](#_Toc160265903)

[Example data 23](#_Toc160265904)

[The output layer 24](#_Toc160265905)

[Input Slices 26](#_Toc160265906)

[Foundational data sets 26](#_Toc160265907)

[Dealing with messy regulations 27](#_Toc160265908)

[Example Transformations 28](#_Toc160265909)

[Output Layer-> report cells 32](#_Toc160265910)

[Overview 32](#_Toc160265911)

[Report Layout description in RegDNA 32](#_Toc160265912)

[Report filter and aggregation in RegDNA 33](#_Toc160265913)

[Example of filters and aggregation in excel example 34](#_Toc160265914)

[Consistent model of computation 35](#_Toc160265915)

[Tests 35](#_Toc160265916)

[Model of computation 36](#_Toc160265917)

[Lineage model 37](#_Toc160265918)

[The importance of lineage 37](#_Toc160265919)

[Attribute lineage – pre-execution of transformations. 37](#_Toc160265920)

[Attribute lineage for single attribute derivation transformations 37](#_Toc160265921)

[Entity to entity lineage for data set transformations 37](#_Toc160265922)

[Entity to attribute lineage for data set transformations 37](#_Toc160265923)

[Attribute lineage for the input slices of generation transformations. 38](#_Toc160265924)

[data lineage -post-execution of transformations 38](#_Toc160265925)

[row lineage 38](#_Toc160265926)

[cell lineage 38](#_Toc160265927)

[Transitive lineage 39](#_Toc160265928)

[Divisible executable lineage 39](#_Toc160265929)

[overview 39](#_Toc160265930)

[Test data creation 40](#_Toc160265931)

[Incremental mapping 40](#_Toc160265932)

[Partial implementation of a reporting framework(s) 40](#_Toc160265933)

[Enemies of divisible lineage 41](#_Toc160265934)

[Technical definition of RegDNA 41](#_Toc160265935)

# Introduction

## Overview

RegDNA is a standard for defining regulatory data models and regulatory logic suitable for statistical and prudential reporting.

## Standards based approach

Standards have the power to transform the way we do things, enabling efficient collaborative approaches for common problems. We can consider the example of how the HTML standard was instrumental in creating the world wide web.HTML was invented by Tim Berners-Lee in 1993, as a standard for making web pages, and he also developed the first simple web browser, which was initially called World Wide Web, but later changed to Next. HTML was first used and became popular within the CERN nuclear research center.Now HTML is managed by the www consortium, a vendor neutral organization, and has inspired the creation of many commercial web browsers such as Chrome, Firefox, Edge, Opera, which have made the standard universal for making web pages.Imagine a world where content creators had to make web pages differently for every browser, this is the situation we have now with regulatory data models and logic.RegDNA is an open standard, accessible to everyone, and governed by the Eclipse Foundation in a vendor neutral way. The Eclipse Foundation oversees over 400 open source projects and specifications, including the Eclipse IDE and the Java Enterprise Edition specification, which are both used by millions of developers.The ‘Eclipse Free BIRD Tools’ project provides a free and open source reference implementation of tools that use the standard, and a platform for running regulatory logic defined in RegDNA.The RegDNA standard is 'software friendly', it is designed to enable and facilitate the development of other tools and platforms based on the standard, free or commercial, by any party, creating an ecosystem of interoperable and exchangeable tools around the standard.

## RegDNA as a simple language

RegDNA is a very simple language. There are free and open tools for editing and navigating the language, and for splitting sentences into components to help create new artifacts such as visualizations or translations to other technologiesIt is easy to see the changes between different versions of regulatory logic or data models written in RegDNA.For more complex logic and calculations, RegDNA uses well-structured placeholders that can be filled with functions in mature programming languages such as Java, Python or C-sharp. These functions are limited to ensure they are simple and focused on specific parts of regulatory logic.RegDNA is designed to be easy to use as a basis for code generation (for example to Java, Python or C-sharp ) so that executable versions run in mature frameworks that are familiar to millions of developers.

## Tools

Tools are like HTML editors for the HTML standard. They help non-technical experts in regulation write text in the RegDNA language, with features such as auto-completion, validation, visualization, source control, and collaboration. They enhance speed, productivity, quality assurance, and collaboration efficiency.RegDNA is designed to make it easy to create tools, by giving easy access to the structure of descriptions written in RegDNA, to enable navigation and visualization.

## Platforms

Platforms are like the web-browsers for HTML. They can run the content to create regulatory reports by executing regulatory logic and calculation on a concrete implementations of regulatory data models (e.g. data models stored as XML files or tables in a relational database)

RegDNA compatible platforms are intended to stop the vendor lock-in problem in regulatory reporting, allowing banks to stay with vendors as long as they are helpful, and giving them a way to switch to another vendor using the same standard, just like someone might change from a Microsoft browser to Firefox or Chrome.

## Ecosystems of platforms, tools, and content

RegDNA enables the creation of an eco-system of compatible tools, platforms, and content. This is partly because of the financial incentive for platform or tools developers to follow the standard. Who would create a browser today that does not support HTML?The open-source Eclipse Free BIRD Tools project ensures there is always one free and open source example of tools and platforms. The license of this example also lets it be used freely as the foundation for other free or commercial tools and platforms so that vendors do not have to start building tools from scratch if they want.

## Software-friendly standards

### Auto-generation of software

RegDNA not only simplifies the creation of artefacts (regulatory data models and logic, but also enables the creation of artefacts that can be used to automatically generate functional platforms.Automatic generation of software artefacts from core definitions is the most common method of software engineering today. No longer do we have to manually create database structures, user interface and database communication, and data-aware forms for each domain concept such as a 'loan'. This means that we can automatically generate functional platform prototypes, instead of developing them from scratch, reducing costs, increasing ROI for commercial platforms, and significantly lowering the maintenance burden associated with handwritten solutions...as we say in the software world, 'what you write yourself you must maintain'.From text in RegDNA we can automatically generate the ‘implementation’ artefacts needed for a technical platform. Following the DNA theme we like to call one description of regulatory logic and datamodels, such as the data models and transformations of the BIRD project, a ‘RegSeed’...or in the case of BIRD a BIRDSeed.Also following the theme we like to call the code/tools to generate a platform a ‘RegPot’...when we combine the RegSeed and the Regpot we can auto generate a RegPlatfrom with the implementation artefacts required. The seed contains all the business information, the pot knows about the technical implementation.

Eclipse Free BIRD Tools currently automatically generates one platform that is desktop based, and stores regulatory data as local files. This platform uses the Java Eclipse Modelling Framework and its Ecore standard to make an implementation based on mature frameworks and components, understood by many developers.The Eclipse modelling framework itself is a framework that takes models and logic defined in the Ecore standards and automatically generates usable components . This is the usual scenario for creating platforms...first generate from RegDNA to another mature standard that itself uses auto-generation to create usable implementations, then use that framework to generate the platform that is well supported by a large number of developers. RegDNA is designed to be similar to common standards that create their software artefacts using auto-generation. This approach of combining a Regseed containing Regulatory datamodels and logic, with a RegPot containing information on how to create a specific technical platform with the necessary services is visualized below:

A tree with roots growing in a pot

Description automatically generated

#### Ecore and the Eclipse modelling Framework

Ecore is a reference implementing OMG's EMoF standard for model driven engineering, and it is very similar to OMG's UML as a modelling standard.The Eclipse Modeling Framework can automatically generate implementation components based on Ecore descriptions, such as storing data in XML files, or making data aware forms for editing data.For example, we can define the concepts of Loan and Counterparty in Ecore, and then automatically generate a simple model-aware editor for loans and counterparties that saves the data in XML files. Ecore’s EOperations let us add executable functions in modern languages like Java, and run them using modern and well understood platforms like the Java Virtual Machine.The reference implementation in Eclipse Free BIRD Tools converts RegDNA into Ecore (which is very easy to do, in about 30 lines of code) , and then uses the Eclipse Modelling Framework to automatically generate visualization, editing, and storage components.

#### Python Django, Java Spring and Microsofts code-first C-sharp framework

3 other excellent candidates for creating platforms are Python Django, Java Spring and Microsoft’s code-first C-sharp framework.

These enable generation from simple description of an artifact (which is very similar to RegDNA’s description of datamodels), the generation of databases, web forms, and their synchronization with intermediate format as Java/Python/C# ‘classes’ which are usable software components where we can add isolated detailed behavior of regulatory logic.

### Pluggable workbenches

If platform components are created from RegDNA, they usually work well together if they use the same mature implementation framework (e.g. they all use Ecore, or all use Django)We can build on this idea and make components that can be plugged in or extended in a 'workbench'...For example the Eclipse desktop IDE, or Microsofts VSCode which can run on the desktop or in the cloud via Github codespaces.These workbenches enable the addition of more components, and support interoperability and re-use of the existing components already available on those workbenches (such as source control and integration with collaboration tools such as Github and GitLab)

## Don’t re-invent the wheel

RegDNA adds to a simplified version of the existing [Ecore](https://en.wikipedia.org/wiki/Eclipse_Modeling_Framework) standard and its [Xcore](https://wiki.eclipse.org/Xcore) text representation. We don’t re-invent the wheel where we don’t need to!

Ecore, as mature open standard for model based engineering, already covers the wide range of data-modelling concerns.

We add an incremental change to this existing standard to enable a simple outline of regulatory logic suitable for statistical and prudential reporting.

## Artefact descriptions

RegDNA is used to define 3 kinds of artefacts. These are data models, transformations, regulatory reports. Transformations represent the business user friendly regulatory logic which is applied to data in the data model to populate the reports.

## Model of computation, lineage

To create a platform, which executes an executable version of regulatory logic, we follow a particular ‘model of computation’ which ensures excellent complete lineage and drilldown of executable logic pre-execution, and excellent lineage and drilldown of calculated numbers post-execution, and also enable performant platforms.

This model of computation can be understood by anyone who understands Excel. RegDNA is designed to facilitate this model of computation to ensure quality, lineage enabled, performant platforms.

Data Models

## The key roles of models in modern software

A common way of designing modern software applications is to use a view, model and store pattern.The view that allows interaction with the model is usually created from the model, and the store that holds the data is also usually created from the model. This way, there is no need to build these things manually, and this approach ensures consistency across all layers.It is often the case that the store is a relational database.The model is often a software representation in a current object oriented language and when we have a database as the data store, the most usual way to connect the 'object oriented' model with the 'relational store' is to use mature 'object relational mapping’ (ORM) frameworks that can turn the data in the database into software artifacts (objects), let them be modified in software by software functions, and then save them back to the database smoothly.ORM frameworks also handle common issues like the differences between vendors' databases, and the migration of existing data when models change which is a key issue in regulatory reporting where new reports might need to run on old data.

These patterns save a lot of time and maintenance, to the point that given a model, functional, basic applications can be produced and reproduced in minutes or hours.

## Different modelling approaches

When we model a domain (such as financial data needed for regulatory reporting) we typically model using entity relationship diagrams, and there are many tools for this. The methods of modelling look fairly similar regardless of the tool or standard used. We can see below a part of the BIRD data model described in the SQLDeveloper tool, and then the ‘Ecore Tools’ tool:

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

However the methods do differ slightly, and also the means of storing models and working with them is different between tools.

## Software focused models vs database focused models

Some modelling tools are more oriented towards relational databases, where the relationships are based on primary and foreign keys, which can also be composite keys of multiple columns.Some modelling tools, such as those that use the class diagrams of the OMG’s Unified Modelling Language UML standard, do not rely so much on relational database concepts, and are often used to model software models.When we use software-focused models, we can better model, and even autogenerate from our tools, the model part of modern software frameworks, from which we can then delegate the tasks of creating and converting and migrating and vendor-specific database creation to modern ORM tools…and automatically create user interfaces and web APIs. This is more difficult to achieve if we only model the relational concepts of the ‘bottom layer’ of modern software frameworks.Having said that, the software-focused and database-focused entity relationship models are not very different, and we can potentially convert between the 2 or try to combine the best of both worlds with annotations. Starting with a software-focused model will speed up the development of usable software than a database-focused model.

### Open modelling standards vs closed modelling standards

Some tools, like Oracle's SQLDeveloper, use common modelling standards, but save them in closed formats that are not documented and limited by licenses.This makes it difficult to extract data from the tool for the purpose of making software applications, unless you want to use other tools from the same vendor. Such closed formats can also make it hard to change to another tool.Tools that use and store model definitions in an open standard, such as UML, Ecore, or RegDNA, allow for easy switching to different tools, easy access to the content made by those tools, and easy creation/extension of open source or commercial tools.

## Reuse of open standards

RegDNA uses the well-established Xcore standard https://wiki.eclipse.org/Xcore for data models, which is a textual way of expressing the Ecore standard https://en.wikipedia.org/wiki/Eclipse\_Modeling\_Framework. These are open, software-oriented, modeling standards.RegDNA also introduces a separate compatible extension for transformations that handle statistical and prudential regulatory operations, as we explain later...because there is no existing thing to use for this purpose.XCore and Ecore have been applied in many situations already and have received years of feedback to ensure that they are suitable for their goals.We simplify XCore a bit by removing any features that are not necessary for us.We also use a slightly modified version of the open source XCore editor in eclipse Free BIRD tools.

## Terminology

In the field of data-modelling there are a lot or terms used that have similar meanings, such as entities, tables and cubes. Another example is attributes, columns, variables. In RegDNA we use the following terms

Entity: e.g Loan is an Entity, entities have a set of attributes.

Attribute- E.g Currency is an Attribute, or balance. Attributes have a type.

Derived Attribute – E.g ‘gross carrying amount’, these are attributes which are defined by an operation which uses other attributes and derived attributes as inputs (such as carrying\_amount+acrued\_interest)..the operation does not have a name, the operation is not re-used across multiple derived attributes, in a real sense the operation is the derived attribute.

Type- E.g String , Int, and Currency\_Domain are types

Enumeration- some Attributes have a type such as Currency\_Domain, which is a named set of allowed values such as ‘USD, EUR, JPY’, we call this set of allowed values an Enumeration

Enumeration Literal- JPY is an example of an enumeration literal.

## Input Layer

We use an Input layer, an Enriched Input Layer and a group of Output Layers to model regulatory modelsThe Input Layer is the basic data of financial operational data (e.g. loans, securities, derivatives, parties) and their properties (e.g. interest rate , nominal amount, carrying amount, accrued interest). We represent these as entities, with attributes, where the attributes will have a type or a list of possible values.

## Enriched Input Layer

The enriched input layer is a superset of the input layer, it includes attributes that are derived from items in the Input layer. For example, it might include ‘gross carrying amount’ which could be calculated by adding together the carrying amount and accrued interest of a loan. We say that such derived amounts are defined by an operation. In practice in RegDNA we only model the enriched input layer, but we can distinguish which are the derived amounts because they are defined as operations with the keyword ‘op’ as shown later.

Not that ‘Derivation Transformations’ are the operations that transform from input layer attributes to Enriched input layer derived attributes.

## Output Layers

Output Layers are set of entities that store information in a style that is close to the submission format for a regulation.

The required submission format for some regulations such as Anacredit, and in future European IREF are dataset based, with lots of granular data. We model these datasets as individual entities, one for each table in the dataset.

The submission format of some regulations is template-based like Corep and Finrep regulations defined by the EBA and European commission. For template-based regulations, we model these as one entity for each template. The entity contains attributes which can be filtered and summed to populate report cells. The format of filters and reports is discussed later, and we provide an example with real data to make this clearer.

Note that ‘Generation Transformations’ transform from input layer attributes and Enriched input layer attributes to Output layer attributes, these are discussed in a later section.

## Datamodel examples in RegDNA

We show here an example here of an BIRD\_EXCHNG\_TRDBL\_DRVTV\_POSTN\_RL entity, taken from the BIRD data model and defined in the RegDNA standard:

A screenshot of a computer

Description automatically generated

We describe the keywords used in the language below.

package: package describes a set of entities, or a set of enumerations, or a set of datatypes . In this case the package name is input\_tables and it describing the set of entities in the enriched input layer of BIRD.

import: some other packages that are used in this package.

annotation: a description of an annotation allowed in this file.

class: An entity, with a name. we choose the keyword class instead of entity because we want to re-use the battle-test3ed Xcore standard, and its editors and tools.

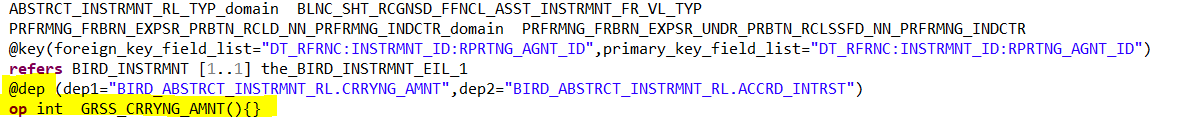
id: identifies one attribute as the uniqueID. Note that there can only be one of these in an entity. The BIRD datamodel currently uses composite keys to identify a unique item, in this case we add a unique\_ID attribute and we set it by convention to be the concatenation of the contents of the primary keys.

@key highlights that we are going to provide an annotation about key information as a set of key/value pairs. Annotations are about the following element in the file. In this case we note the extra information that in the published BIRD data model BYR\_PRTY\_ID is part of a primary key, and also that it acts as part of a foreign key.

refers: describes a relationship, this is an association between 2 entities , and equivalent to what is called ‘association’ in UML class diagrams. In database-focused modelling standards relationships are usually defined by foreign keys links to other tables primary key (single or composite)…where this is the case in BIRD we store that information in an annotation about the foreign and primary key items involved in the relationship between to 2 entities.

[1..1] describes the cardinality of an association, the first number is the lower bound, the second is the upper bound. We use -1 to mean ‘many’. So [1..1] means we have exactly one associated item, [0..-1] would mean we have zero to many associated items.

We show below an attribute from another entity also that is a derived attribute, and is therefore described as an op(eration). The details of the operation are not shown (this is left to the generated platform to describe in a modern language like Java or Python) but we do describe what are the dependant fields used in the operation using a, @dep annotation, in this case BIRD\_ABSTRCT\_INSTRMNT\_RL.CRRYNG\_AMNT and BIRD\_ABSTRCT\_INSTRMNT\_RL.ACCRD\_INTRST… Note that these are described with the entity name, then a ‘.’ then the attribute name,



Note that for each attribute or operation we state the type (like String or int) or the enumeration (e.g. PRDNTL\_PRTFL\_domain) the types are described in another package as below:

A screenshot of a computer

Description automatically generated

The enumerations are described in another file, as follows..note that each value has a name, a code and an associated number:

A screenshot of a computer screen

Description automatically generated

and use the enum keyword to describe enumeration. Language based tooling can help automatically navigate between enums used to describe the types of attributes, and the list of values represented by an enum.

## Version differences

Note that since the language format is based on straight forward text (and not XML or JSON or a binary format) then showing the difference between 2 version with standard text ‘diff’ tools gives a clear indication of what has changed.

## Logical datamodels and disjoint subtyping

BIRD contains a logical datamodel and a relational model. The logical model contains a concept of ‘inheritance’ or ‘subtyping’ which is very easy to describe as inheritance in Ecore /UML/RegDNA. It also contains a concept of ‘disjoint subtyping’ to describe concepts that group together entities from multiple hierarchies (known as arcs in SQLDeveloper). We describe this same concept of disjoint subtyping in Ecore/UML as a ‘delegate’, this is a pattern sometimes known as ‘preferring aggregation over inheritance’, we show a typical comparison here.

In SQL developer we show here that Instrument table has 2 ‘arcs’ which contain sublclasses. The right hand arc is ‘is instrument type by origin ‘ containing subtypes ‘Instrument resulting directly from a Financial contract’ and ‘Instrument resulting directly from a Credit Facility’ and the right hand is ‘instrument type by product’ containing ‘Over the counter (OTC) Derivative instrument’ , ‘Off-balance instrument’ , ‘Securities financing transaction (SFT)’ and 2 others not in this screenshot. In a simplified manner we can say that we are going to need to choose one subtype from each arc when we represent an instrument. We show the 2 arcs here in blue, and then show refined diagrams showing the content of each arc:

A computer diagram with green text

Description automatically generated with medium confidence

Contents of ‘Intsrument by Product’ Arc:

A screenshot of a computer screen

Description automatically generated

Contents of ‘Instrument by Origin’ Arc:

A screenshot of a diagram

Description automatically generated

In ECore, and therefore in RegDNA we describe the same as

A screenshot of a computer screen

Description automatically generated

Note that the arrows black diamonds refer to a ‘containment’ relationship in Ecore, which is also known as a ‘composition relationship’ in UML, and the lines with white arrow heads are inheritance/subtyping. This allows the same case as in disjoint subtyping where we must choose a subtype from each delegate.

Note that in RegDNA (as in XCore) containment relationships are described using the keyword ‘contains’ and inheritance using the keywork ‘extends’ …when we have an entity that exists only to be a supertype we use the abstract keyword to depict this, as in:

A yellow and black text

Description automatically generated

A black text on a white background

Description automatically generated

A white background with black text

Description automatically generated

# Logical Transformations

## Introduction

Using RegDNA we can present a way of defining the rules that are needed to change data stored in a data model into a format that can be used for reporting, such as report templates or reporting datasets. We want to do this in a way that is simple for business users to understand without knowing technical details, and also in a way that can help in making transformations that can be executed on data. We call these business-friendly transformation ‘logical transformations’. We want anyone who can use basic excel to be able to understand the transformations.The transformation should allow tracing of items from the reporting format back to data model attributes from specific entities, we call this attribute lineage. The transformations should also allow making executable transformations that can trace the data items (actual numbers produced by running the transformations in the reportable information back to the original data in the data model, we call this data lineage...this 30 second video shows a visual difference between attribute lineage and data lineage.https://youtu.be/YVgpGv\_PK4U

## Derivation transformations

Derivation transformations describe how to attributes in the enriched input layer should be calculated from attributes in the input layer and/or other attributes in the enriched input layer.

## Derivations that are single attribute enrichments

### overview

A common case of derived attributes in simpler regulations like Finrep and Anacredit are those that take items from one entity in the input layer, and produce another attribute in the same entity.

### example

An example of this would be gross\_carrying\_amount, which could be described as carrying\_amount + accrued\_interest . We describe these attributes using the op keyword. If we wish we may give an example in a language, using ‘< >’ brackets describing the language, although this is ignored. An example is show below using python, If we generate from this a platform that is python based, we could use those contents.

Here is an example from an entity called BIRD\_ABSTRCT\_INSTRMNT\_RL which happens to have a reference to the BIRD\_INSTRMNT entity

A computer screen shot of a computer code

Description automatically generated

### dependencies

We describe the lineage as 'dep' annotations (dep for dependencies). If we generate from this example platform that is python-based, we would use the dependencies in the Python function.

It is conceivable that we can have a process that discovers the dependencies by inspecting the code in a language such as Python or Java, this processing would be slightly different depending upon the formal language used in the platform.

### Platform model of computation

Note that for single attribute enrichments, the generated platform will provide a function in some useful language like Java/Python/C# ..in most some cases this can be generated, in others it needs to be hand written in the specific placeholders functions generated.

#### *Limit access to limit complexity*

Note that for single column enrichments, the generated platform will have a function that takes the other **attributes** from the entity and creates the value for a derived **attribute.** Note importantly that thefunction processes only other attributes of the same entity, or attributes that are part of an entity that has an association relationship with it, so in this case we could use attributes from the related BIRD\_INSTRMNT table in the operation if we wanted to since we **refer** to it.

#### side effect free function

Note that the function does not 'set' any values of attributes , it is just producing the value for this one derived attribute, we say it is side effect free.

#### immutable results

Note that the derived value is only set by this function, no other processing can change it. We say it is immutable since it does not change.

These properties of limited scope, side effect free functions, and immutable results help us with lineage, limiting complexity, and allowing performance (together these properties also make it easy to parallelize, cache, and perform only functions we need, as we see in the popular platforms for data processing such as the Apache Spark platform for common within the banking industry.

Derivations that are not single attribute enrichmentOverview

For regulatory processing it is common for more complex prudential regulations like COREP to require the derivation of data where we are not simply deriving one attribute for one entity, but rather netting, matching, or splitting whole sets of data to produce new sets of data. Importantly the produced dataset might have a 1 to N , or Not 1 , or N to N relationship with the rows of the source dataset(s)

### Examples from Corep

We highlight below some examples from COREP , where it is very common that processing logic described by regulations has multiple steps , each of which can be considered as producing a whole new dataset. Some examples are:

**Credit risk:** the matching of exposures with collaterals, resulting in covered and uncovered parts of exposures

**Credit risk mitigation:** the creation of inflows and outflows of risk between exposure classes through substitution of a collateral's risk weight

**Counterparty risk:** netting of derivative transactions and margins into netting sets in counterparty risk

**Market risk ladder approaches**: the creation of artificial notional positions for derivatives in market risk

**Market risk FRTB sensitivities-based approach:** the aggregation/netting of interest rate sensitivities within a grouping ‘bucket’ such as currency based buckets

**Market risk FRTB sensitivities-based approach:** the creation of correlation pairs of netted sensitivities **within** a grouping bucket,

**Market risk: FRTB sensitivities-based approach:** the creation of correlation pairs of net values **between** buckets,

**Liquidity reporting:** the creation of related cashflows.

These processes result in cases where the information aggregated and put into report template cells looks very different than the original input layer in form and importantly with totally different cardinalities. For example, one loan might create multiple inflows/outflows or covered/uncovered exposures. One derivative transaction might result in many notional positions, many derivatives might result in one netting set.

### Representation in RegDNA

We represent this case as the derivation of derived entity.

For example in FRTB SBA we need to net sensitivities by risk factor as an intermediate step to get items that we can use in report cells in the C91 report.

As a naming convention we like to use \_derived at the end of the entity name.The refers sections in these entities always refer to the entities that were used to create this derived entity.

### Example

In this example we have a derived entity called Netted\_delta\_sensitivities\_per\_risk\_factor\_and\_tenor\_derived which is derived from the entities called Delta\_sensitivity and GIRR\_Risk\_factor.

A screenshot of a computer program

Description automatically generated

In the generated platform we will have an operation that takes those populated source entities and creates instances of the derived entity.

That operation will create this data set with just the refers associations set.

This is quite subtle, note that all the derived attribute operation work once the refers associations are set.

So creating a derived entity is a 2 step process, first create the table with the referred associations set. And then apply the derived attribute operations.

Note that the referred associations may have a cardinality greater than 1, so for example **refers** Delta\_sensitivity [ 0 .. -1 ] in the Netted\_delta\_sensitivities\_per\_risk\_factor\_and\_tenor\_derived entity, may refer to more than one row in the source sensitivities table….these n links will be set in the platform function…

This also means that operations in this case will not just be adding items from one row, but might be aggregating items from the linked rows in the source data sets.

The operation to create the dataset (with just refer’s associations set) may need to access a particular subset of the attributes/operation in the source tables, and we highlight these in the dependency annotation of the entity.

In our example we needed "Delta\_sensitivity.Risk\_factor\_Identifier","Delta\_sensitivity.Tenor",

and "Risk\_factor.Risk\_factor\_Identifier" to ensure we can create the correct reference links.

Note that all attributes of a derived entity are derived attributes and so will be set by operations as described above, although there is one exception to this. This exception is as follows: we may choose to set an attribute that is not a refers association in the operation creating the derived entity. This helps us in rare cases like ordering, indexing of items when we create multiple rows in the derived entity for one row in the source entity, we do this because it is the only time we can do it, as some information is lost after the first step of creating the refers items.

Note that like single attribute operation, these operation to create the links are also side effect free, immutable and access limited.

### Example of Lineage

Here is an example of lineage in derived entities. We see that operations can be considered as excel formula, we have used the ‘show precedents’ feature in excel to point to arrows to the source cells of a formula.

A screenshot of a computer

Description automatically generated

You can see that one netted sensitivity in row 17 is taking values from 2 rows (7 and 8) we can say that rows 7 and 8 are source rows for …in fact all derived items in row17 are taking as input items from the rows 7 and 8, or from row 17…all operations in one row, take inputs from the same ‘source rows’

Note that we can have entites built from entities built from entities, representing the multiple dependent steps that are common in common in COREP processing.Note specifically that they a follow a pattern of being data set to dataset transformations, which is a very common model in modern analytical platforms, enabling detailed lineage of processing steps.

## Generation transformations

Generation transformations collect data from the input layer and enriched input layer and put it in the shape of an output layer. Note that later we can translate data from an output layer to a report template for template-based reports, we have one output layer per report template for template-based regulations.

Generation transformations do not calculate new results as this is the responsibility of the derivation transformations, it is just collecting existing results. We make the assumption that all the attribute names used in the output layer, are also used in the input layer and enriched input layer.

There is a 2 stage approach for generations transformations in RegDNA, first split/join the input layer data into ‘input slices’, then transform per attribute.

### Example data

We have attached here csv files following the BIRD 6.3 format providing an example of a loan to a central bank , a loan to a household party with a guarantee, an OTC derivative, and a non-negotiable bond. For enumerated values we have provided long names along with the codes . this data is attached as a zip file called ‘input data.zip’

### The output layer

In this example we have a transformation to an output layer entity for an F5.01 Finrep report.

Note that for template-based regulations we choose to have an output layer entity per report, and not merge these into 1 output layer for many reports. This helps with divisible lineage, as described later in the lineage section.

We note that there is a process in BIRD that determined what are the attributes that the F05.01 output layer should have using the reference set of attributes names in the BIRD input layer and enriched input layer. This is the ‘mapping’ process in BIRD, but the process is not of relevance to RegDNA, we re only interested in the resulting output layer entity described using attributes and enums that are also used in the of the input layer.

The input to BIRD’s mapping process is the details shown in the Finrep F05.01 annotated template here.

A screenshot of a computer

Description automatically generated

The dimensions of that annotated template are shown below

|  |
| --- |
| Accounting portfolio (APL) |
| Metric (ATY) |
| Base (BAS) |
| Counterparty sector (CPS) |
| Entity id (Entity\_Id) |
| Instrument (MCB) |
| Main category of collateral or guarantee received (MCG) |
| Main category (MCY) |
| Purpose (PUR) |

The BIRD mapping process finds the related set of attributes using the attribute names and enumerations of the input layer. These attributes (known as reference variables in BIRD) are shown in the diagram below, these make up the attributes of the output layer entity for the F05.01 report.

We plan to populate this output layer entity with many rows, so that we can filter according to enumerated attributes, and then aggregate numeric attributes such as carrying amount, so as to populate report cells (we describe the population of report cells in a later section).

We describe how to get data from input layer entity rows into output layer entity rows in a RHS ->LHS format, but what does the RHS look like :

A blue arrow pointing to a blue line

Description automatically generated

Further more …we can ask what does 1 of these rows in an output layer entity represent? Is it a loan ? an asset? An exposure?

Let us consider the report cells in a Finrep report F01.01, that are the result of an aggregation of rows of an output layer entity.

A screenshot of a computer

Description automatically generated

We can see that since the cells are aggregations of rows in the output layer entity, that the individual rows can be many things, loans or derivatives, or debt securities for example, so how do we translate these different things to get information into the output layer entity for the report?

Q.) How to Translate multiple things to the output layer entity?

A.) One by One…first loans and advances , then debt securities, etc

### Input Slices

The different financial products (like loans and advances, derivatives, securities) have different properties, and are treated in different ways by regulations. In the input layer these can be from different entities, with different entities linked to them by different conditions…for examples loans might be linked to a counterparty as main debtors, they might have an associated credit facility, or securities might be linked to a party as an issuer. Furthermore securities might be linked to a rating, although we don’t care about that rating for Finrep, or at least for F05.01 report.

To deal with this separate treatment for separate types of financial product we split the input layer into a set of parts relevant to Finrep processing, we call these parts ‘input slices’. Since the description of Finrep regulation is often differentiated on different ‘main categories’, this is the way we choose to split the input layer for Finrep reports..note that for market risk in COREP we would split a different way (probably by asset class) and for COREP credit risk yet another way (perhaps by exposure classes which have different treatments in the COREP credit risk regulation).

### Foundational data sets

We note that that a common approach in regulatory reporting solutions is to translate from a normalized data models (like the BIRD input layer) with many entities and relationships, into a single ‘flat’ normalized structure with many attributes. These flat structures are sometimes called ‘record types’ or ‘foundational data sets’.

These foundational data sets can be filtered by attributes such as ‘currency’ or ‘held\_for\_sale’ to get the set of rows relevant to report cell, or report row or report column. Then numeric values such as ‘carrying amount’ or ‘notional amount’ can be summed across these rows to get aggregated amounts to populate the cells in the reporting templates of template-based regulations.

#### Foundational data sets per regulation

We note that when such foundation sets are truly flat it is common to have one per regulation or part of regulation, this is because different levels of information are needed for different regulations.

Take the example of one derivative transaction. In Finrep the foundational dataset might need one row per transaction, so that we can get its nominal amount. In liquidity reporting we might need one row per associated cashflow, so we might need many ‘flat’ rows for one derivative transaction, and a different set of attributes would be important (like inflow/outflow direction). For COREP counterparty risk, we need the exposure amount for a single netting set, and one netting set will exist covering multiple derivative transactions. For COREP Market risk we will need 2 ‘notional positions’ for a single derivative transaction in the maturity ladder approach with their derived notional, or multiple sensitivities for a single derivative transaction in the FRTB Sensitivity based approach each with their numeric delta sensitivities.

We note also that not all banks will use a single datamodel for all regulations. They might not need to report COREP market risk tempaltes because they have a small trading book, or because they have internal models that bypass that part of regulation. They may have a system for COREP counterparty risk that they love, and so use a completely sperate model and system for those reports. We do not want to force banks to use part of the data model that they do not need, by including those attributes in the description of foundational datasets, for example if they do not use FRTB SBA approach for COREP market risk then we do not want to force them to populate FRTB related data such as delta sensitivities which are hard to obtain, so that they can have these in foundational datasets.

#### Foundation sets per report?

It is less common, but we may even choose to have a foundational data set per report. Each with a different structure. This means more structures, and that some structures are similar, but it has many advantages, in particular it enables divisible lineage which is a very powerful way to save time and money for banks, as we describe later in this document.

We take this approach of one foundation data set per report in RegDNA. Interestingly this means that a foundation dataset for a report is just the output layer entity for the report, since that is a flat representation of the attributes needed for the report. It also means that the output layer is the union of multiple input slices relevant to the report, and each slice contains only attributes and entities relevant to that report. For example we do not link ratings information in the slice, if the report does not need ratings information, so we do not force users to populate ratings information if they do not use it for the reports they are working with. Also, we do not include derivatives information in any slice for a report if the related report does not deal with derivatives.

### Dealing with messy regulations

As an example form BIRD, we note that in Finrep when we consider reports for ‘Loans and advances’, these include relevant instruments stored in the BIRD\_INSTR entity in the BIRD data model, but it also includes non -negotiable debt securities in the BIRD\_SCRTY\_PSTN entity. This is because non-negotiable debt securities must be treated as loans according to the Finrep regulation. This is one of the many ‘messy parts’ of regulation. The regulation does not care that we are trying to put things into nice clean boxes and grids!

Any standard for describing regulatory logic must be capable of dealing with the messy parts, and that is certainly true in RegDNA. In this case we create 2 separate ‘input slices’ for such reports, both of which will be used in generation rules for reports such as F05.01 which is a report about loans and advances. One input slice contains non-negotiable debt securities, the other contains products traditionally classed as loans and advances. When we describe the per-attribute transformations for the Finrep F05.01 report, it is always very clear which slice we are using, which adds clarity, especially since different input slices might use the same entity and attributes for different purposes.

### Example Transformations

Before we describe fully the generation transformations we show an example of generation rules for F05.01 described using RegDNA, using the BIRD input layer data model as input. You will see that there is 2 input slices with a main entity and filter, and these have lhs->rhs ‘per attribute transformations’, with the attributes matching the attributes above

A screen shot of a computer

Description automatically generated

#### Output layer

The generation rules are stored as one file per output layer, which is one per report for template-based regulations like Finrep. The output layer for the generation transformation is highlighted with the ‘outputLayerEntity’ keyword.

#### Main entity

We use the mainEntity keyword to state the ‘main entity’

The ‘main entity’ in generation rules is an entity where 1 row produced in the output layer entity will have 1 related row in the ‘main entity’ and not more than one row. For example for an input slice that relates to loans and advances, this would be the instrument entity in BIRD which contains things like the id of a loan, it would not be the party entity , we don’t have one loan per party. Note that this does not mean we have one row in the output layer entity for every row in the main entity….for example the instrument table in BIRD must be filtered to get just the items relevant to the loans and advances items (excluding other things like OTC derivatives which also exist in the instrument entity)

These linked tables are not actually explicitly defined but looking at the LHS part

of the transformation, we see which linked entities are used.

#### Input Slice

Under the ‘main entity’, we list all input slices with that main entity, for that report. The input slice has a name given after the inputSlice keyword. The input slice contains a list of per-column transformations which we describe in a section below.

#### Linked entities

Not all the information that we require for the per-attribute transformations of an input slice is available in the related main entity. There will be a few entities from which we need to get extra information, like BIRD\_PRTY for a loan counteparty’s institutional sector, or ‘BIRD\_ABSTRCT\_INSTRMNT\_RL’ for the carrying amount of the loan…note that there might be technical tables that help with finding the related rows in those tables like ‘BIRD\_INSTRMNT\_ENTTY\_RL\_ASSGNMNT’, and we do not include these in linked entities, but they will eventually be used in the technical execution of the ‘hidden join’. Note that there might be 2 parties for a loan (maybe due to a joint loan) but we might only be interested one (e.g. the main debtor), in this case it is the responsibility the hidden join to choose the correct one so that it produces a flat structure.

#### The hidden join

This is not described and is left as a technical aspect of the generated platform to deal with. This is to make sure that the input slices can be understood by business users without being overwhelmed by the details of how the technical join happens. The technical join will be defined in the language of the platform….it is best defined by technical users.

##### Example of the hidden join using the example data

In this file ‘generation\_transforamtions\_example.xlsx’, derived from the example input data in input\_data.zip, we see examples of the hidden joins for the loans and advances input slice of F05.01and Non-Negotiable bonds input slice of F5.01. These are in the tabs named ‘loans and advances joined ‘ and ‘Non-Negotiable bonds joined’ . You will see that all the linked entities for a input slice are shown in the csv file. The join produces a ‘flat’ structure. We see that we show in the flat result where each attribute came from by using the original attribute names in the second row and the name of the entity from which they came from in the first row, e.g.

A screenshot of a computer

Description automatically generated

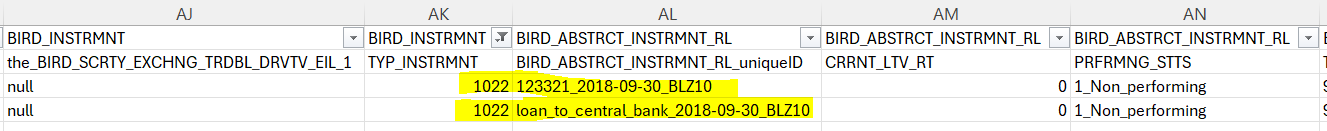
#### Filter

Each input slice has an optional filter. The filter gives a view of how we filter the main entity and its linked entities to get just the subset of data that we need. It focusses on describing the filters on the main entity but can filter on linked entities also. We do not include technical filters needed here to make the flattening happen, this is the responsibility of the hidden join.

Often we re-use filters, and input slice names, within a reporting framework like Finrep, but this is not enforced. We might see ‘Loans and Advances’ input slice name, with the same filter, but different per-column transformations, in the generation rules for different Finrep reports.

##### Example of the filter using the example data

We show an example of how the filter is applied, we can see this in the tabs Loan\_and\_Advances\_Filtered, and Non\_Negotiable\_bonds\_Filtered of the generations\_transformation\_example.xlsx file. Note that the filter removed the OTC derivative row and the guarantee the in Loans\_and\_Advances input slice, since it does not meet the filter (TYP\_INSTRMNT =5 or TYP\_INSTRMNT=14 is not in the filter).



#### Per-attribute transformations

To generate rows in the Output Layer entity, for each relevant input slice, we define for one row from the input slice (which includes attributes of the main entity, and might include attributes from the linked entities) and say which attribute from the input slice is used to populate each attribute in the row of the output layer entity. We show again here the example of per-attribute-transformations from one of the ‘Loans and advances’ input slice for the F05.01 report.

A close-up of a computer screen

Description automatically generated

Note that for each per-attribute transformation the left hand side will be one attribute (or derived attribute) from the input slice (i.e. from its main entity or a linked entity), and it will have the same type as the right hand side attribute from the output layer entity even if that type is an enumeration. Note that we refer to attributes with qualified names, so we note the package, entity, and attribute separated by dots. Where the left-hand side attribute is a derived attribute we use the keyword selectDerivedAttribute, if it is not a derived attribute then we use the keyword selectAttribute.

Not that this approach provides proof and confidence that the input layer and enriched input layer are complete enough to populate the output layer because we know that each attribute exists in a specific place relevant for that particular input slice, and the attributes that are needed together are all accessible together (e.g. we can access the carrying amount and the held\_for\_sale indicater from the same input slice relevant to ‘loans and advances’) so this enables the filters and aggregations needed to populate the report as described later.

##### The special case of defaulting.

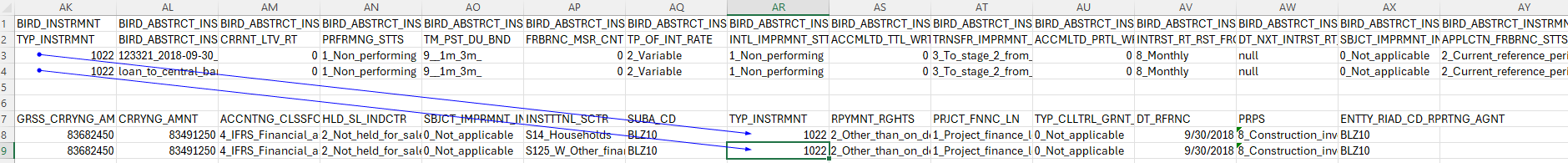
We also have a special case where we might just set the attribute of the row in the output layer as a particular value like “1” in which case we use the keywork **selectValue** , or we might wish to set it to a particular enumeration literal for an attribute with an enumeration as its type, in which case we use **selectMember** keyword,…in this case we ensure that the enum literal is a valid literal of the enumeration for that attribute of the output layer.

##### Example of per attribute transformation with example data.

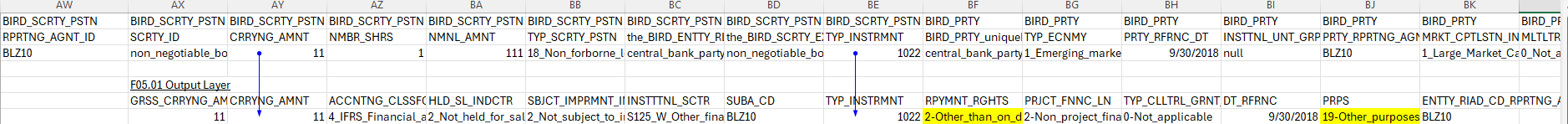
In the example spread sheet , in the loans\_and \_advances\_transformed tab we see at the top the filtered loans and advances again (just a repeat from the loans\_and\_advances\_filtered\_tab), and at the bottom (which is post-transformation) we see that we have one row for each row in that filtered dataset, and one column for each attribute in the output layer entity.

We have used Excel formula for each cell to just retrieve the item mentioned in the per-attribute transformation.

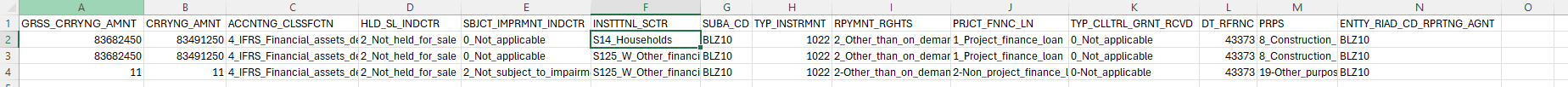
Using Excels feature to visualize links between Excel formula’s inputs and outputs (‘show precedents’ ) we visualize the data lineage in Excel for some of the cells.



Note that in the non\_negotiable\_bond\_transformed tab we do the same for non-negotiable bond, For the special case of defaulting, we have just set the cell to a specific value(we highlight these in yellow



and in the union\_of\_transformations tab, we just take the 1 resulting row from the transformed\_non\_negotiable\_bond tab and put it below the 2 resulting rows from the transformed\_loans\_and \_advances tab. This represents our populated output layer as below



For some regulations (like IREF and Anacredit) we can stop processing when we produce the output layer entities, as this represents the submittable format. In template based regulations like Finrep , we need a further step to create the values for the report cells from each output layer entity. This step is a fairly simple applications of detailed filters on selected values from enumerated values (such as held\_for\_sale\_indicator = held\_for\_sale), and the aggregation of numeric fields (such as carrying amount)..We will show in the next section how we can easily create the report cell.

## Output Layer-> report cells

### Overview

The required submission format of some regulations such as Anacredit, and in future European IREF are dataset based, with lots of granular data. The submission format for some other regulations is template based like EBAs Corep and Finrep. We describe here how we describe the report structures, and also how we populate report cells from the contents of the output layer entity for a report.

We describe the report layout and the ‘output layer to report cell’ logic in one file per report template.

### Report Layout description in RegDNA

The first section of the file describes the rows and columns (we will add sheets in future also), we show an example here, the keywords highlighted in red are hopefully quite clear and simple to understand, we also mention after the **outputLayer** keyword the name of the output layer entity….with tool support for the language this will be validated immediately to see if it is a valid entity, and in fact will usually allow for auto-completion if you type the first few characters

A screenshot of a computer program

Description automatically generated

### Report filter and aggregation in RegDNA

We describe how the report cells are to be populated by describing for each report cell a set of filters on the rows of the output layer, and which numeric value should be summed.

It may be more efficient in future to define the filters per row and per column and sheet, and so the filters for a cell would be the combination of those filters. We cannot describe all cells like this because some are derived by formula, such as the ratio of 2 cells…however in that case we could assume any formula described in the report simply override the row and column and sheet level filters.

For each cell we describe the cell with an optional ID, described after the keyword **datapointID**, and highlight which row and column it is for. We then describe, after the metric keyword, the numeric attribute that we will be summing (GRS\_CRRYNG\_AMT in the example below.) We then describe the filters to that we plan to apply in the filter sections.

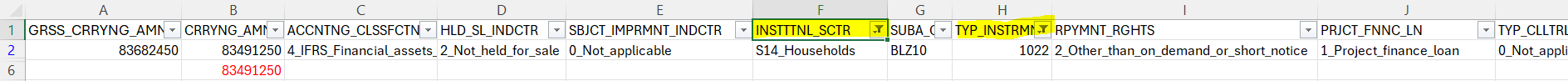
A screenshot of a computer program

Description automatically generated

Note that for filter we do not have a shorthand to say ‘all enumerated literal of an attribute’ or a particular subset. We do not have the concept of hierarchies for literals of an enumeration, this is because we wish to keep things simple and usable by common software frameworks that do auto-generation of datastores and visual frontends/web APIs.

### Example of filters and aggregation in excel example

We can show this as an example of spreadsheet filter. It in tab F05\_01\_filtered int e example spreadsheet. Here we have just used the Excel filter functionality to filter on the required attributes of the output layer entity, and then used the sum functionality of excel for the metric (we show the result in red). In this case we have applied the filters for the cell in row0060 and col0060 in this example, and summed the relevant metric which is carrying \_amount. Note that we filtered out anything that did not have an institutional sector of S14\_household.



### Consistent model of computation

Note that we can still consider this ‘output layer entity -> report cell’ process as an example of many individual dataset to dataset transformations (one for each cell), with lineage to the source rows (that met the filters for that cell), however in this case our target dataset has one column and one row (i.e. it is one cell)

# Tests

Since banks have the responsibility for the final reports that are submitted to regulators, banks usually do tests with specific data to test specific regulatory logic.

RegDNA does not provide a specific way to describe test data, however when a platform is generated this will give an implementation method to store input layer data (e.g. in XML files in the reference implementation or most likely in relational databases in other platforms).

Since platforms likely use auto-generation they will have a way to create data-aware forms that understand the types of attributes and the allowed enumerated values. Here is an example from the reference implementation.

A screenshot of a computer

Description automatically generated

We see on the bottom right here a panel that knows what attributes are needed for the BIRD Instrument entity, the types for each attribute (e.g. text or numeric or date) and what are the allowed literals for enumerated attributes (such as the dropdown that you can see for the allowed currencies)

It is often forgotten that that test are often better written before transformations as acceptance criteria, specifying a clear input and expected output pair that encapsulates real regulatory expertise in bitesize sharable artefact.

# Model of computation

As described in the transformation section, we see that platforms generated from RegDNA files will follow a model of computation, which can be easily visualized by users of Excel, and can be summarized as meeting the following properties:

* Immutable dataset to dataset transformations
* Operations that translate from attributes to a single attribute
* Immutable operations and transformations
* Side effect free operations and transformations
* Limits on complexity

# Lineage model

## The importance of lineage

For anyone involved in regulatory reporting it is a life spent mostly determining what is the long line of processing to determine how a number should or did get into a box. If we can just show the lines that saves a lot of time and effort!

RegDNA has an unrivalled lineage model, which has been a design goal from its very inception and never compromised on. It enables attribute lineage, data lineage, transitive lineage and divisible lineage as described below

## Attribute lineage – pre-execution of transformations.

This is the lineage of how attributes are derived from other attributes, such as carrying amount

### Attribute lineage for single attribute derivation transformations

This is the lineage of how attributes are derived from other attributes, such as carrying amount example.

### Entity to entity lineage for data set transformations

This is the lineage of how an entity is derived from one or more entities such as the Netted\_delta\_sensitivities\_per\_risk\_factor\_and\_tenor\_derived being derived from the entities called Delta\_sensitivity and GIRR\_Risk\_factor.

### Entity to attribute lineage for data set transformations

This is the subtle example of which attributes are required to select the correct rows in the creation of a derived entity, such as the requirement to know the In our example we needed "Delta\_sensitivity.Risk\_factor\_Identifier","Delta\_sensitivity.Tenor",and "Risk\_factor.Risk\_factor\_Identifier" to ensure we can create the correct reference links in Netted\_delta\_sensitivities\_per\_risk\_factor\_and\_tenor\_derived

### Attribute lineage for the input slices of generation transformations.

A screenshot of a computer

Description automatically generated

## data lineage -post-execution of transformations

### row lineage

After we have executed out transformations

A screenshot of a computer

Description automatically generated

### cell lineage

From the row lineage described above and the attribute lineage of derivation transformation rules we can create a cell lineage, as show here

A screenshot of a computer

Description automatically generated

video link to attribute vs data lineage https://youtu.be/YVgpGv\_PK4U

## Transitive lineage

Transitive lineage allows out work backwords from report cell, or an output layer entity, and see the subset in input layer attributes (or numbers) that were required by ‘following the lines’

## Divisible executable lineage

### overview

RegDNA allows for divisible attribute lineage, this is built into the design of RegDNA as a primary driver, and is not commonly available in regulatory solutions, despite its ability to save huge amounts of time.

Divisible attribute lineage helps us to understand or debug logic for a report or report cell by throwing away hundreds of items and operations that are not needed for that task so we can see the wood for the trees. Divisible data lineage also provides a neat way to focus on just the calculations and intermediate calculated values that were used in the calculation of a report or report cell.

We highlight this with an old example from an earlier revision of RegDNA. We see in this example that we greyed-out anything not required via transitive lineage for one of the report cells to show the concept of divisible attribute lineage.

![divisiable](<https://github.com/eclipse/efbt/blob/develop/org.eclipse.efbt.wiki_images/spec_images/DD_Vismode.PNG>)

Divisible lineage enables efficient test data creation, incremental mapping, and maintenance, and simplifies partial implementation of a reporting framework or set of frameworks as described below

### Test data creation

When we create test data for a report, or even a report cell, it is useful to know what is the small subset of input data that is required from the input layer, to populate just that report or report cell.

This allows efficient creation of test data as we don’t need to think carefully about hundreds of attributes that are just not needed in the calculations for that report. Also, by using just the subset of attributes used, this means that the test don’t go quickly out of date as the data models and transformation changes…it will only be out of date when the particular formula and transformations for that test changes, and when the data model changes in a way that changes any one of the small subset of attributes.

### Incremental mapping

With divisible lineage we can take an incremental approach to data mapping from a banks data to the regulatory data model. So at first a bank can map just the data from their ultimate operational systems into the subset of fields in the input layer required for 1 report. The input layer fields required by that 1 report can be found using lineage to get the subset of fields required for report 1. This report can be checked and reviewed and tested before incrementally moving on to the next report(s) and therefore mapping further fields.

A common finding when banks try to map all required data as a “big bang” exercise is that they try to map everything before running the logic, and then on the first run of the transformation logic to populate all reports, almost all reports fail to be populated correctly , or the report contents are mostly unexpected/empty results. Do deal with this complex situation a lengthy and expensive debug process follows for the next number of months, often with a set of consultant experts in the system used.

### Partial implementation of a reporting framework(s)

It is important to note that not all banks report all regulations and all approaches for each regulation, and that they may even use different systems for some regulations if they are confident in existing systems. So a bank might report COREP but use the maturity ladder approach for market risk and not FRTB for market risk and so use a different set of report templates. Or they might use an internal model for all of market risk, or they might have a small trading book and therefore have a waiver so they don’t need to report market risk, or they might have spent so much time invested in an existing product from COREP market risk that they decide to do that part of COREP with a separate system. For example it can be quite common that market risk departments look after market risk processing, and other departments look after other parts of COREP, potentially using/choosing different systems.

Since we have transitive and divisible attribute lineage, we can find the subset of fields required for a bank’s subset of reporting requirements…there is no point in a bank providing all the input layer fields required for reports to a data model if they don’t need to submit those reports using the datamodel.

### Enemies of divisible lineage

Some design decisions make divisible lineage very difficult, and we avoid these in RegDNA.

#### Massive flat output layers

output layers with lots of columns can give problems, partly as they require fully complete joins for reports to get fields like ratings, even for reports that don’t need those fields.

#### ‘forward thinking’

😊 Well that is a joke, but thinking about regulatory systems from the input layer, and how we can transform in full steps forwards towards output layers can result in processing that makes attributes in intermediate tables mandatory, even when they are only needed for some reports or some report cells. Thinking instead of how we work back from output layers for a report, towards the input layer makes us focus on introducing the smallest number of dependencies.

# Technical definition of RegDNA

How is the RegDNA standard defined? The Regdna.xtext file (https://github.com/eclipse/efbt/blob/develop/regdna/dsl/org.eclipse.efbt.regdna.dsl.parent/org.eclipse.efbt.regdna.dsl/src/org/eclipse/efbt/regdna/dsl/Regdna.xtext ) defines a grammar which refers to the data model via the regDNA.ecore file ( https://github.com/eclipse/efbt/blob/develop/regdna/eclipse\_model/plugins/org.eclipse.efbt.regdna.model/model/regdna.ecore ) …amazingly just these 2 quite short files together are enough to autogenerate most of the Eclipse Free BIRD Tools reference implementation of RegDNA including its smart navigable text editors, code generators, and data aware forms.