Work Package 2: Workflow

Capturing Pharmacometrics Workflow Concepts

Related task of the project (Task # and full name):	D6.17
Author:	Jonathan Chard
Approved by:	Justin Wilkins

1. Definitions

1. Acronyms and Abbreviations

Term	Definition
PROV-O	A standard developed by the W3C for capturing information about provenance, encapsulating *entities, activities* and *agents* involved in the creation of digital artefacts For more information see the official documentation at https://www.w3.org/TR/2013/REC-prov-o-20130430/
PROV-N	A human readable implementation of the PROV-O standard, which defines statements that should be used to capture provenance information
Thoughtflow	A term coined within DDMoRe to define the combination of *workflow* and the rationale behind the decisions taken during a modelling project

Contents

1. Definitions 2
1. Acronyms and Abbreviations 2
1 Introduction 6
1.1 Purpose 6
1.2 Audience 6
1.3 Scope 6
2 PROV-O Terms 7
2.1 Entity 7
2.2 Activity 7
2.3 Agent 8
2.4 Relationships 8
2.4.1 Activity - Entity relationships 8
2.4.2 Entity - Entity relationships 10
2.4.3 Activity - Activity relationships 12
2.4.4 Entities - Agent relationships 12
2.4.5 Activity – agent relationships 13
2.4.6 Object - object relationships 13
3 Pharmacometrics Workflow Concepts 14
3.1 Entities 14
3.1.1 Typical project entities 14
3.1.2 Assumptions 15
3.1.3 Decisions 16
3.2 Relationships between entities 16
3.3 Activities 17
3.4 Agents 18
4 Mappings onto Provenance Concepts 19
4.1 Conventions 19
4.1.1 Naming conventions 19
4.1.2 Entity creation 19
4.2 Entity capture 19
4.2.1 Usage of existing attributes 20

4.2.2 Entity Types 20
4.2.3 Entity Attributes 21
4.2.4 Assumptions 22
4.3 Entity Relationships 23
4.3.1 Incorporating Activities into a relationship 24
4.3.2 Describing Entities 24
4.3.3 Relationship Metadata 25
4.4 Activities 25
4.4.1 Activity capture 26
4.5 Agents 28
4.5.1 Person Agents 28
4.5.2 Software Agents 28
4.5.3 Environment Agents 28
4.6 Modelling Steps 29
5 User actions to capture 30
5.1 Principles 30
5.2 Scenarios 30
5.2.1 A “message” is sent to the Thoughtflow server 30
5.2.2 Create child model 32
5.2.3 Weak links between models 32
5.2.4 Updating a file 33
5.2.5 Moving a file 33
5.2.6 Adding metadata to an Entity 34
5.2.7 Updating the description of an Entity 35
5.2.8 QC Status 37
5.2.9 Making an Assumption/Decision 38
5.2.10 Updating Entities revisited 38
5.2.11 Template models 39
6 Querying the Workflow Store 41
6.1 Storing information in the Workflow store 41
6.2 Obtaining the model development tree 41
6.3 Getting Entities 41
6.4 Getting Activities 41
7 Solution Design 43

7.1 Provenance Concepts 43
7.1.1 Identifiers 43
7.1.2 Entity IDs 43
7.1.3 Activity IDs 44
7.1.4 Bundles 44
7.2 Components 44
7.2.1 Version Control System 44
7.2.2 Provenance Infrastructure 45
7.2.3 Thoughtflow Repository 47
7.2.4 R package 48
7.2.5 Task Execution Service 48

1 Introduction

1.1 Purpose

Workflow is centred on the recording and exploration of provenance: information about entities, activities, and agents involved in producing a piece of data – or any other kind of entity – and the relationships between them.

The PROV-O ontology (http://www.w3.org/TR/prov-o/) provides the basis of our approach to defining and capturing provenance information. PROV-O can supply the necessary framework to invalidate entities and/or activities, allowing the effects of making a change in an analysis to be assessed based on the relationships of the affected entity or activity with downstream activities and entities: everything dependent on the change could, for example, be flagged as invalid, or as requiring reassessment.

The purpose of this document is to map PROV-O terms onto pharmacometrics workflow concepts, to allow the capture of the day to day activities performed within a typical pharmacometric project

1.2 Audience

The Audience of this document is:

- Pharmacometricians, who can contribute to the completeness of the mapping between workflow concepts and PROV-O
- Developers, who will be responsible for implementing the specification

1.3 Scope

This document covers:

- An analysis of PROV-O
- An analysis of pharmacometrics concepts
- A specification for mapping pharmacometrics concepts onto PROV-O, and possible extensions
- Messages that encapsulate user actions and concepts to be stored within a pharmacometrics project

The document does not provide an exhaustive overview of PROV-O – a detailed description can be found at https://www.w3.org/TR/2013/NOTE-prov-overview-20130430/

2 PROV-O Terms

PROV-O defines the following terms that are used to capture all the information necessary to define the provenance of items within a document.

2.1 Entity

An entity is a physical, digital, conceptual, or other kind of thing with some fixed aspects; entities may be real or imaginary.

It has the following “subtypes”:

- Collection
  - A grouping of Entities, and only entities.
- Plan
  - A plan is an entity that represents a set of actions or steps intended by one or more agents to achieve some goals
- Bundle
  - A bundle is a set of provenance descriptions, so it can contain Entities, Activities and Agents
  - Note however that a bundle cannot contain more bundles

2.2 Activity

An activity is something that occurs over a period of time and acts upon or with entities; it may include consuming, processing, transforming, modifying, relocating, using, or generating entities.

An **activity**, has:

- *id*: an identifier for an activity;
- *startTime*: an OPTIONAL time (st) for the start of the activity;
- *endTime*: an OPTIONAL time (et) for the end of the activity;
- *attributes*: an OPTIONAL set of attribute-value pairs representing additional information about this activity.

An example activity statement in PROV-N would be:

```
  activity(a1, 2008-08-30T01:45:36, 2008-08-30T01:45:36.123Z,
[prov:type=”ddmore:commit”])
```

2.3 Agent

An agent is something that bears some form of responsibility for an activity taking place, for the existence of an entity, or for another agent's activity.

Subtypes:

- Organisation (e.g. Leiden, Pfizer)
- Person (e.g. a User)
- Software Agent (e.g. R, Monolix)

2.4 Relationships

PROV-O defines a set of relationships that describe the interactions between each of the entity types above.

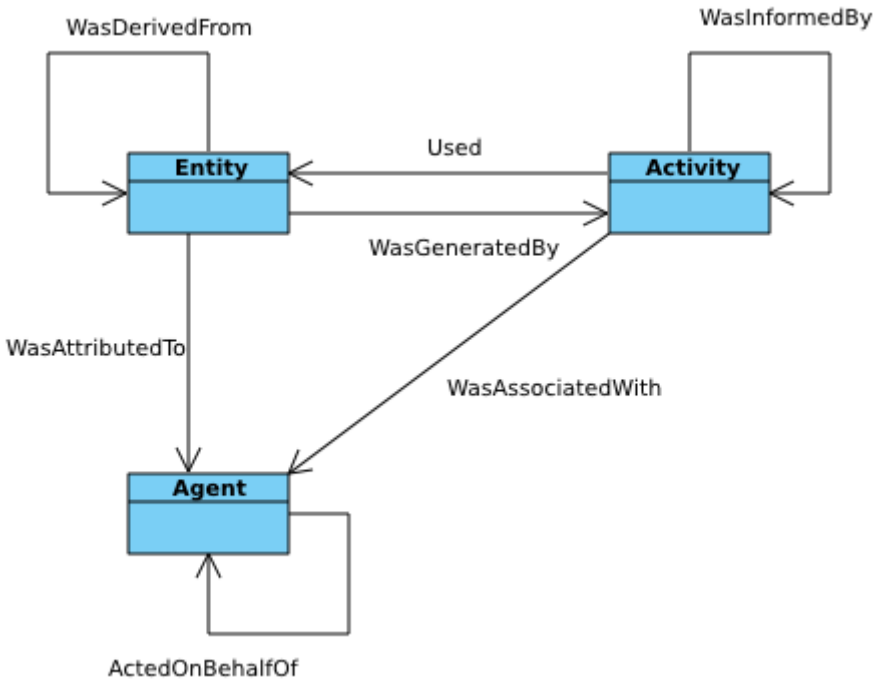


Figure 1: Top level interactions[1]

2.4.1 Activity - Entity relationships

The following diagram lists the possible relationships that can exist between an activity and an entity.

[1] Original image from https://www.w3.org/TR/2013/REC-prov-dm-20130430/#prov-core-structures-top

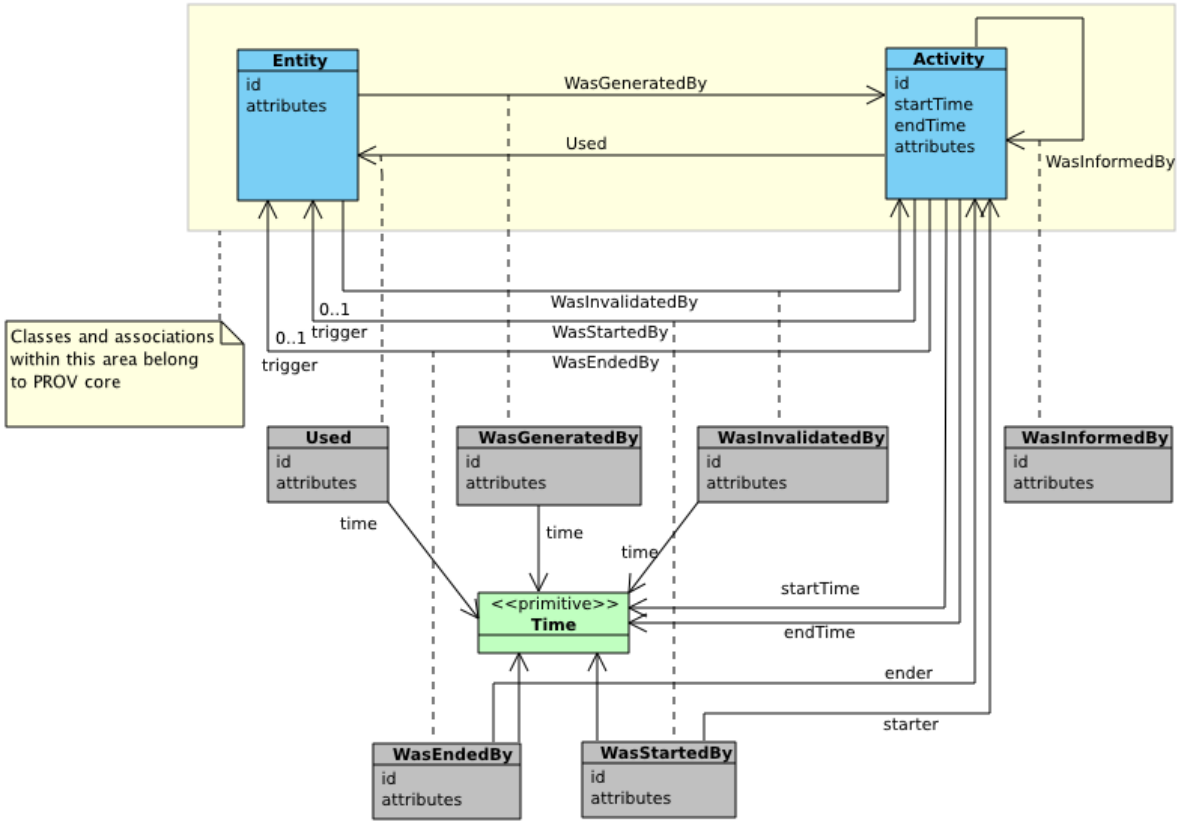


Figure 2: Entity-Activity Relationships[2]

The most relevant to us are:

Was generated by	Generation is the completion of production of a new entity by an activity. This entity did not exist before generation and becomes available for usage after this generation.
Used	Usage is the beginning of utilizing an entity by an activity. Before usage, the activity had not begun to utilize this entity and could not have been affected by the entity
Was invalidated by	Invalidation is the start of the destruction, cessation, or expiry of an existing entity by an activity. The entity is no longer available for use (or further invalidation) after invalidation.
Was started by	**Start**◊ is when an activity is deemed to have been started by an entity, known as **trigger**◊. The activity did not exist before its start. Any

[2] See original image at https://www.w3.org/TR/2013/REC-prov-dm-20130430/#figure-component1

	usage, generation, or invalidation involving an activity follows the activity's start. A start may refer to a trigger entity that set off the activity, or to an activity, known as **starter**, that generated the trigger.
Was ended by	**End** is when an activity is deemed to have been ended by an entity, known as **trigger**. The activity no longer exists after its end. Any usage, generation, or invalidation involving an activity precedes the activity's end. An end may refer to a trigger entity that terminated the activity, or to an activity, known as **ender** that generated the trigger.

2.4.2 Entity - Entity relationships

The following diagram indicates in more detail the relationships that can exist between two entities:

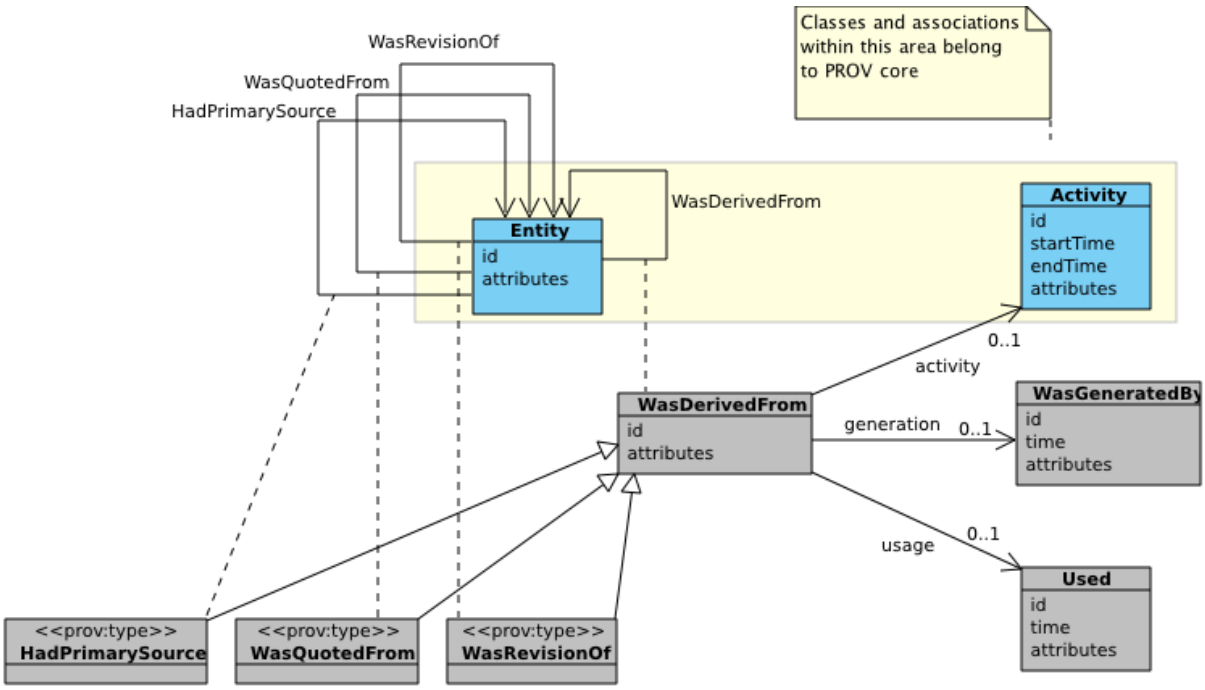


Figure 3: Derivations[3]

There is one type of relationship that can exist between entities ("derived from"), which has three subtypes that we can use:

Term	Strict definition
Derived	A derivation is a transformation of an entity into another, an update of

[3] See original image at https://www.w3.org/TR/2013/REC-prov-dm-20130430/#figure-component2

from	an entity resulting in a new one, or the construction of a new entity based on a pre-existing entity.
Revision	A revision is a derivation for which the resulting entity is a revised version of some original. The implication here is that the resulting entity contains substantial content from the original. Revision is a particular case of derivation.
Quotation	A quotation is the repeat of (some or all of) an entity, such as text or image, by someone who may or may not be its original author. Quotation is a particular case of derivation.
Primary Source	A primary source for a topic refers to something produced by some agent with direct experience and knowledge about the topic, at the time of the topic's study, without benefit from hindsight. Because of the directness of primary sources, they 'speak for themselves' in ways that cannot be captured through the filter of secondary sources. As such, it is important for secondary sources to reference those primary sources from which they were derived, so that their reliability can be investigated. A primary source relation is a particular case of derivation of secondary materials from their primary sources. It is recognized that the determination of primary sources can be up to interpretation, and should be done according to conventions accepted within the application's domain

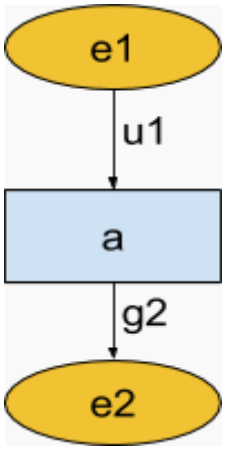
Derivations are specified as below:

```
wasDerivedFrom(ex:d; e2, e1, a, g2, u1, [ex:comment="a
righteous derivation"])
```

Here:

- d is the optional derivation identifier, e2 is the identifier for the entity being derived,
- e1 is the identifier of the entity from which e2 is derived,
- a is the optional identifier of the activity which used/generated the entities,
- g2 is the optional identifier of the generation,
- u1 is the optional identifier of the usage, and
- [ex:comment="a righteous derivation"] is a list of optional attributes. In PROV-N these fields are used to capture the type of derivation, i.e.:
  - revision - prov:type='prov:Revision'
  - quotation - prov:type='prov:Quotation'
  - primary source - prov:type='prov:PrimarySource'

In this way we can capture the activity that generated the derivation.



2.4.3 Activity - Activity relationships

It is also possible to have relationships between activities.

These are summarized below:

Was informed by	Some anonymous entity passes between two activities. So, the prov:wasInformedBy property allows the construction of provenance chains comprising only Activities
Acted on behalf of	Delegation is the assignment of authority and responsibility to an agent (by itself or by another agent) to carry out a specific activity as a delegate or representative, while the agent it acts on behalf of retains some responsibility for the outcome of the delegated work.

2.4.4 Entities - Agent relationships

The following table lists the types of relationship that can exist between an agent and an entity

Attributed to	Attribution is the ascribing of an entity to an agent

The “attributedTo” statement can be used to ascribe of an entity to an agent.

When an entity e is attributed to agent ag, entity e was generated by some unspecified activity that in turn was associated to agent ag. Thus, this relation is useful when the activity is not known, or irrelevant.

An *attribution* relation, written wasAttributedTo(id; e, ag, attrs) in PROV-N, has:

- *id*: an OPTIONAL identifier for the relation;
- *entity*: an entity identifier (e);
- *agent*: the identifier (ag) of the agent whom the entity is ascribed to, and therefore bears some responsibility for its existence;

- *attributes*: an OPTIONAL set (attributes) of attribute-value pairs representing additional information about this attribution.

An example of this is shown below:

```
wasAttributedTo(tr:WD-prov-dm-20111215,        ex:Paolo,        [
prov:type="editorship" ])
```

2.4.5 Activity – agent relationships

The following table lists the types of relationship that can exist between an agent and an activity.

Associated with	An activity association is an assignment of responsibility to an agent for an activity, indicating that the agent had a role in the activity. It further allows for a plan to be specified, which is the plan intended by the agent to achieve some goals in the context of this activity.

An activity **association** is an assignment of responsibility to an agent for an activity, indicating that the agent had a role in the activity. It further allows for a plan to be specified, which is the plan intended by the agent to achieve some goals in the context of this activity.

An ***association*** ◊, written `wasAssociatedWith(id; a, ag, pl, attrs)` in PROV-N, has:

- *id*: an OPTIONAL identifier for the association between an activity and an agent;
- *activity*: an identifier (a) for the activity;
- *agent*: an OPTIONAL identifier (ag) for the agent associated with the activity;

While each of *id*, *agent*, *plan*, and *attributes* is OPTIONAL, at least one of them MUST be present.

A **plan** is an entity that represents a set of actions or steps intended by one or more agents to achieve some goals. The type of a Plan entity is denoted by **prov:Plan**.

2.4.6 Object - object relationships

All the above relationships are subtypes of one relationship - “wasInfluencedBy”. This can be used to ascribe any relationship between any objects.

Influence is “the capacity of an entity, activity, or agent to have an effect on the character, development, or behaviour of another by means of usage, start, end, generation, invalidation, communication, derivation, attribution, association, or delegation”

The W3C recommends not to use this relationship, but we can use it if necessary - though we should attach appropriate attributes to the relationship to specify it properly.

3 Pharmacometrics Workflow Concepts

3.1 Entities

According to the PROV-O specification, an entity is “a physical, digital, conceptual, or other kind of thing with some fixed aspects; entities may be real or imaginary.”

When an “entity” is defined, it normally does not represent the entire entity itself, but indicates where it can be located or retrieved later, i.e. it is a representation of that entity.

3.1.1 Typical project entities

This definition works perfectly on entities that can be directly mapped to files within a pharmacometric project, for instance:

- a dataset
- a NONMEM control file
- an R script
- an output file (e.g. generated by NONMEM or Monolix)
- a graphical output (e.g. a PNG)
- a report (e.g. a Word document or a PDF)

These types of entity can always be retrieved from a secure storage location (e.g. a version control system) at any point in the future, subject to proper backup and recovery procedures, as long as the ID of the entity can be traced back to the individual file (and version of that file).

This definition works less well on “imaginary” or “conceptual” entities as they do not “exist” (as much as anything digital exists), and cannot easily be retrieved at a later date.

Examples of “imaginary” or “conceptual” entities would be:

- a decision
- an assumption
- a data/model/parameter/task object defined within an MDL file

Capturing this type of entity will require creation of a file that represents that entity and contains information pertinent to it, which is added to the version control system.

There are a number of other states/flags that we wish to be able to attach to entities, to identify:

- Importance/Status of an output
- QC status
- “Significance” within the project

These properties can change over time and have a lifecycle that is independent of the entity itself, even though the entity has not changed. Consequently it is not an appropriate use of PROV-O properties to capture this information.

The following information must be supported:

Field	Meaning	Possible values
QC Status	Whether this entities has passed or failed a QC process	<blank> (indicates it has not been QCed) true (passed QC) false (failed QC)
final	This is the final model	true / false
base	This is the base model	true / false
pivotal	This model is pivotal	true / false

Strictly speaking, the *description* of an entity is unrelated to its *provenance*.

However, there must be a link between the description of the entity, and the entity that it is describing, so that the relationship can be followed.

3.1.2 Assumptions

Transparency in the setting and evaluation of assumptions that may impact model application is of great importance in the planning and documentation of any model-informed drug discovery and development (MID3) activity.

Assumptions are documented using a structured ASCII text file, using fields as defined below:

Field name	Meaning	Possible values
Type	The classification of the assumption	pharmacological / physiological / disease / data / mathematical/statistical
AssumptionBody	The assumption itself	Free text (typically 1-2 sentences)
Justification	The justification for making the assumption	Free text (typically 1-2 sentences)
Established	Is the assumption new, or has it been previously established?	new / established
Testable	Is the assumption testable?	true / false
TestApproach	How to test the impact of the assumption	Free text (typically 1-2 sentences)
TestOutcome	How to evaluate the outcome of the testing of the assumption	Free text (typically 1-2 sentences)

An example in XML:

```
<?xml version="1.0"?>
<Assumption>
      <Type>Pharmacological</Type>
      <AssumptionBody>Emax model fixed to 100% is a more
physiological description of the data compared to a linear
model.</AssumptionBody>
      <Justification>Emax model is not better than linear
model; however, for this drug class, Emax of 100% is more
realistic.</Justification>
      <Established>New</Established>
      <Testable>Testable with a wider range of concentrations
(external/future study).</Testable>
      <TestApproach>
Comparison of simulated metrics of interest between the two
competing models.
      </TestApproach>
      <TestOutcome>To achieve a 90&#37; response (assumed to be
clinically meaningful) requires a twofold higher dose using
the Emax model compared to the linear model.</TestOutcome>
</Assumption>
```

3.1.3 Decisions

Decisions (model selection, based on outputs from assumption testing, and similar) are crucial to document. Decisions are defined as entities which physically take the form of simple ASCII text files containing 1-2 sentences describing the decision made.

```
<?xml version="1.0"?>
<Decision>
  This is the base model to be carried forward into the
stepwise covariate analysis.
</Decision>
```

3.2 Relationships between entities

Models, scripts, and data are created and change over time. We wish to capture the relationships between these types of entities so that we understand where “they came from”.

We need to capture the following relationships between “entities” in the system.

Entity type	Types of relationship
Model file	Revisions of a model (i.e. there is a new version of the file that contains the model definition) One model is “derived” from another model, as it contains

	modifications to a previous model that alters how it fits/describes the input data. One model is “influenced by” a model, as there is some loose relationship between them.
R Script	The software agent runs the script; the R script defines what are files are inputs, and how to generate the outputs. The script is not directly responsible for creating the outputs; this is undertaken by the software that runs the script.
Assumption	An assumption influenced the path of development of a model, or the way that a script was implemented.
Decision	A decision is based on a number of entities. The decision itself is created as a result of the act of “taking a decision”.
Description	A “description” adds extra information or context about another entity. It in some way characterizes that other entity. A description can be updated over time, independently of the entity it is describing.

3.3 Activities

An “activity” is something that occurs over a period of time and acts upon or with entities; it may include consuming, processing, transforming, modifying, relocating, using, or generating entities.

In a typical pharmacometrics modelling project, the following are examples of activities undertaken by the modeller that we would wish to capture

- cloning a model
- updating a model
- updating a script
- performing a parameter estimation
- performing a simulation
- performing an SCM
- QC’ing a model
- making an assumption
- taking a decision
- updating an entity’s description (i.e. the metadata that describes an entity within the workflow system)

If an action results in the existence of a new entity in the system, then it should be captured as an activity

3.4 Agents

An agent is something that bears some form of responsibility for an activity taking place, for the existence of an entity, or for another agent's activity.

The PROV-O standard supports the following types of “Agent”:

- A person
- An organisation
- An instance of a software

There is no mechanism to transform Agents and capture revisions. If we attach any sort of state to a user (e.g. “disabled”, or “member of a team”), or alter it in any way, we cannot refer back to the previous revision.

During the life of an analysis project, the follow concepts hold “responsibility” for performing the activities:

- Individuals – i.e. people
- A software package – e.g. R, NONMEM, Monolix
- The environment/platform upon which that software was executed

The first element is important to capture so that we have a record of who undertook the action. The second and third elements are necessary, so that it is possible to reliably reproduce results, and re-execute activities.

As with Entities, the id of an Agent should uniquely and perpetually identify that Agent within the scope of the workflow server. Agents must be valid across different workflow “instances” – i.e. user “jchard” must refer to the same user, in every modelling project.

Likewise, a software agent named “R-3.1.2” must always refer to the same software instance, to guarantee the same results when an activity is rerun. This includes, in this case, the combination of packages and versions that may be used by that “instance”.

In order to fully capture the information regarding the software used to run an activity, we need to store:

- The software and version used
- The host platform/environment used on which that software was installed

We therefore propose an additional type of Agent – “environment” – which defines the host platform.

4 Mappings onto Provenance Concepts

4.1 Conventions

4.1.1 Naming conventions

There is some inconsistency when naming concepts in PROV-O with respect to capitalisation. Sometimes concepts use Camel Case ("`wasGeneratedBy`", "`wasAssociatedWith`"), and sometimes it uses strict capitalisation ("`prov:Person`", "`prov:Plan`")

We shall use Camel Case when defining attribute names and values.

4.1.2 Entity creation

There are two mechanisms in PROV-O for attaching the responsibility for the creation of an entity to an individual. These are:

- "`wasAttributedTo`". This is a simple Entity –> Agent relationship (see 2.4.4) that indicates that the Agent was responsible for the Entity
- Via the more complex "Agent -> Activity -> Entity" relationship
    - The User Agent was associated with an Activity
    - The Activity generated an Entity

Our convention will be to use the second form to ascribe attribution to an Entity. This allows us to:

- Provide inputs to the Activity.
- Attach attributes onto the Activity
- Invalidate other Entities as a result of this Activity

..as necessary.

By following this convention we avoid confusion around when it is possible to simply use "`wasAttributedTo`" or not.

4.2 Entity capture

The provenance ontology standard requires a minimal amount of information to define entities - all that is necessary is an id, which can be used to universally, uniquely identify that entity. It is also possible to attach any arbitrary attributes to an entity to describe it within the system.

The entity id can also be prefixed with a namespace that can be combined with the ID to further identify it.

The ontology also predefines a number of attributes that may be used to define an entity:

- prov:label (0 or more)
- prov:location (0 or more)
- prov:type (0 or more)

- prov:value (0 or 1)

4.2.1 Usage of existing attributes

4.2.1.1 Namespace

We will use the “namespace” to define the URL of the repository in which the entity is stored.

We will use the “repo” as the shorthand name of the namespace within each provenance document.

Each provenance document will include the following namespaces:

```
default <http://ddmore.eu/workflow/#>
prefix ddmore <http://ddmore.eu/workflow/#>
prefix mid3 <http://ddmore.eu/mid3/#>
```

There will be extra namespaces added that correspond to the type of repository that the code is being stored in, and the unique identifier for that particular repository.

These namespaces are:

Namespace name	Description	Examples
vcs	The type of repository	https://www.github.com/#
repo	The location of the repository	https://github.com/johndoe/examplerepo.git/#

How a client utilises this information is implementation dependent.

4.2.2 Entity Types

The following table summarizes the types of entity we will track in the system that are in addition to the existing types defined by the PROV-O standard. These will be captured using the “prov:type” attribute.

Model	NONMEM control file (.ctl, .mod) MDL file (.mdl) Monolix model (.mlxtran)	prov:type=ddmore:model
Dataset	CSV (.csv) Table file (.tab) Data file (.dat)	prov:type = ddmore:dataset

PharmML archive	.phex	prov:type = ddmore:phex
Standard output object	.so	prov:type = ddmore:so
Output	NONMEM output file (.lst, .out) Monolix output file Output tables	prov:type = ddmore:output
Image	PDF image PNG image JPG image	prov:type = ddmore:image
Assumption	Structured TXT document	prov:type = ddmore:assumption
Decision	Structured TXT document	prov:type = ddmore:decision
Document	HTML document DOCX document PDF document RTF document	prov:type = ddmore:document
Description	A description of another entity	prov:type = ddmore:description
QC status	An entity that encapsulates the QC status of another Entity	prov:type = ddmore:qc
Bundle	A Bundle that defines the contents of a commit to a version control system	ddmore:type=ddmore:commit
Bundle	A Bundle that defines the Description of an Entity	ddmore:type = ddmore:description

4.2.3 Entity Attributes

“prov:location” will be used to identify the location within the version control system of the file that this entity represents.

“prov:label” is designed to “provide a human-readable representation of an instance of a PROV-DM type or relation”. A number of labels can be attached.

Therefore we will use “prov:label” to provide a human readable description of the entity or activity, typically the name of the file with the path stripped from it.

4.2.4 Assumptions

An assumption is realised in the repository as an XML file. This allows the user interface to read and display the assumption in an attractive format.

The PROV-O entity will refer to this file in the repository.

It will be necessary to query the Thoughtflow server for “assumption” entities, according to the type of assumption. In order to support this, the following fields will be used as attributes of the Entity. Note that these will be namespaced to “mid3”:

Metadata field name	Possible values
mid3:assumptionType	pharmacological physiological disease data mathematical
mid3:established	new established
mid3:testable	true false

These fields *must* be identical to the content of the XML file and should be calculated and added onto the entity by the server when the entity is processed.

There will be an activity - “make assumption” – that has one output – the assumption itself. There may be a number of inputs that provide some reasoning for the assumption that can be tracked. The activity will be linked to a user agent who was responsible for making that assumption.

In terms of the impact that this assumption has on the project, the “wasInfluencedBy” relationship is the most general relationship that we can use for this purpose. We will make this relationship more specific by attaching the “prov:type = ddmore:predicates” onto the influence, as follows:

```
entity(repo:abc123/assumptions/assumption1.xml,
[prov:location=”assumptions/assumption1.xml”,
prov:type=”ddmore:assumption”, mid3:testable=”true”])
entity(repo:bda321/models/run1.mod, [
prov:type=”ddmore:model”])
wasInfluencedBy(repo:bda321/models/run1.mod,
repo:abc123/assumptions/assumption1.xml,
[prov:type=”ddmore:predicates”])
```

Should this assumption change, or become invalidated, then the “influenced” model needs to be examined to establish the impact of this change.

This relationship is unidirectional; if the model changes, there is no impact on the assumption. However, there is still an influence on the later version of the model that needs to be followed.

4.3 Entity Relationships

Models, scripts, and data are created and change over time. We wish to capture the relationships between these types of entities so that we understand where “they came from”.

We have 4 types of “derivation” available to us. Note however that we can attach metadata to a relationship in order to impart extra meaning.

The following relationships are available:

- “derived from” - is a transformation of an entity into another, an update of an entity resulting in a new one, or the construction of a new entity based on a pre-existing entity
- “revision of” - a derivation for which the resulting entity is a revised version of some original
- “quotation” - a quotation is the repeat of (some or all of) an entity, by someone who may or may not be its original author.
- “primary source” - A primary source for a topic refers to something produced by some agent with direct experience and knowledge about the topic, at the time of the topic's study, without benefit from hindsight.

An extra relationship is also available - “wasInfluencedBy”. This is a general relationship that can be used to capture a link between **any object in the system** (i.e. between agents, entities and activities). All other relationships (e.g. used, generated, derived from, etc.) are specialisations of this relationship.

We will use these relationships as follows:

Relationship	Usage
derived from	Capturing parent/child relationships between models
revision of	Capturing updates/new versions of existing files. This replaces the previous version of the entity (i.e. the previous version is no longer “in” the project)
quotation	Not planned to be used, although it may be useful for QC purposes as there is no change to the original. However, semantically, “quotation” does not appear to be the correct term to use for this.

primary source	Would be used when a model is imported into a project from a model library.
was influenced by	A weak relationship exists between two entities; a change in the "influencer" will not necessarily affect the "influencee"

4.3.1 Incorporating Activities into a relationship

In section 4.1.2 we set out the convention of capturing the creation of Entities, whereby an Activity is also included in the relationship between two Entities.

When one Entity is derived from another Entity, "was derived from" relationship allows an Activity to be included in the statement, as below:

```
wasDerivedFrom(generatedEntity, usedEntity, activity,
generatedId, usedId, [prov:type="ddmore:specialisation"])
```

(See section for 2.4.2 details)

In PROV-N, there is no special statement that differentiates between "wasDerivedFrom", "revision" or "primary source". These specialisations of "derived from" are encoded in the prov:type attribute.

The following types of Activity should be used with the different types of derivation:

Derivation type	Activity type
ddmore:specialisation	ddmore:clone
prov:Revision	ddmore:commit
prov:PrimarySource	ddmore:import

This activity can also be used to mark "invalidation". For instance, an output will be invalidated by the action made by a user to an upstream file (e.g. modification of the base model)

4.3.2 Describing Entities

According to section 3.1.1 there is a requirement to attach descriptions to Entities that are captured within the Thoughtflow server. This is not strictly within the scope of "provenance" but it is within the field of responsibility of the Thoughtflow server.

Therefore we need some strategy for incorporating this information into the Thoughtflow database by extending the relationships available in the Prov-O standard.

In Provenance terms, we have two separate entities that we would like to capture:

- The Entity being described (e.g. a model)
- The description of that Entity

Both of these Entities have their own life cycle - both the model and the description can be modified independently, by different users.

The “description” Entity may also incorporate many facets, such as:

- Whether the model is a base model or a final model
- Whether the model has been QC’ed

We will use the “influence” relationship to capture the relationship between the “Entity” and the “Entity Description”, and give the relationship a type of “describes”.

See section 5.2.6 for details on how this is used in practice.

4.3.3 Relationship Metadata

The following table thus summarises the relationship types that we will use to provide extra information about the meaning of the relationship.

Relationship	Attributes	Meaning
wasDerivedFrom	prov:type = ddmore:specialisation	Used to denote that a model is a child of another model.
wasInfluencedBy	prov:type = ddmore:predicates	Used to denote that an Entity is influenced by an Assumption
wasInfluencedBy	prov:type = ddmore:describes	Used to denote that an Entity is influenced by a Description

4.4 Activities

The system must support the following “other” types of Activity (beyond updating or cloning an existing entity)

Metadata field name	Value	Meaning
prov:type	ddmore:commit	Commit changes to the version control system
prov:type	ddmore:estimate	Perform a parameter estimation
prov:type	ddmore:simulate	Run a simulation
prov:type	ddmore:qc	Perform a QC
prov:type	ddmore:decision	Make a decision

prov:type	ddmore:assumption	Make an assumption
prov:type	ddmore:describe	Attach a description to an entity

Every entity creation/transformation ***must*** be linked via an Activity. This includes:

- Making an assumption or decision
- Copying/Moving a file
- Cloning a model (i.e. create a child)
- Updating metadata

This allows us to capture who performed the action, when it took place, and add any other metadata as necessary.

It also allows us to record the *impact* of this change elsewhere in the system, as Entities are only *Invalidated* by Activities.

4.4.1 Activity capture

Activities should be captured as follows:

- Create the activity
- Record the inputs and outputs with “used” and “wasGeneratedBy” statements
- Record the plan and software agent with “wasAssociatedWith”
- Record who ran the activity with “wasAssociatedWith”

4.4.1.1 Plans

A “plan” is a special type of Entity that makes up the set of actions or steps that was followed by the Agent to achieve their goals in the context of an activity.

In terms of the types of activities that take place in a project, the “plans” in each case are:

Activity Type	Plan
Run an R script in batch	The R script that was executed
Run an estimation with NONMEM	N/A
Run an estimation with PsN	The PsN script (“execute”)
Run an scm with PsN	The PsN script (“scm”)

4.4.1.2 Associating an activity with a plan

Consider an R script being run. We need to capture:

- The software used to run the script

- The script that was run
- The inputs to the script, and the artefacts that the script generated
- The user who ran the script

This can be achieved with the following statements:

```
entity(R-3.1.2, [prov:type="prov:SoftwareAgent"])
entity(userA, [prov:type="prov:Person"])
entity(RScript, [prov:type='prov:Plan',
prov:location='/path/to/file/Script.R'])
activity(RunRScript, -, - )
wasAssociatedWith(RunRScript, R-3.1.2, RScript)
wasAssociatedWith(RunRScript, userA)
```

Where a1 is the activity, ag1 is the agent, e1 is the plan, followed the attributes.

The first "wasAssociatedWith" statement links the activity with the software and script (plan). The second "wasAssociatedWith" statement links the activity with the user.

4.4.1.3 Links between activities

The main mechanism for capturing links between activities is the "wasInformedBy" relationship. This indicates the "some anonymous entity" passed between the two activities; we don't need to capture entities passing between them. The intent of this is to capture chains of activities.

This can be used to capture instances where some script spawns new activities, for instance:

- An R script calls "estimate"
- PsN generates multiple NONMEM runs

In both of these cases, we do not know what may have been passed to the "estimate" function as an input – the model could have been constructed in memory, and supplied to the called routine.

Likewise, the artefacts generated by the downstream activity may or may not be used by the calling script. The information we can capture is:

- The original script
- The inputs to that script
- The activity spawned by the script
- The artefacts generated by the spawned activity (we do not necessarily know the inputs)
- The outputs generated by the original script

We can be more exact about the outputs generated by the original script; it is

"all outputs" – "outputs generated by spawned activities"

For the purposes of "regeneration", all that should be done is run the original script, which will execute the same downstream activities as before. However, for the

purposes of “correctness”, we should record which activities generated which entities.

4.5 Agents

The “agent” itself is described by an id and a list of attributes. We will specify the list of attributes we will use to describe each agent.

4.5.1 Person Agents

The following information should be used to define person agents:

Attribute name	Description
prov:label	Human readable description of the user
username	A unique user name for this user. Typically used to authenticate this user within the organisation e.g. the LDAP name
email	The user’s email address

4.5.2 Software Agents

The following information should be used to define person agents:

Attribute name	Description
prov:label	Human readable description of the software
version	Version number of the software
name	The name of the software
type	Any type information to attach to the software e.g. test, production

4.5.3 Environment Agents

An “environment” agent will be denoted with a prov:type of “ddmore:environment”, i.e.

```
agent(ex:ag4, [ prov:type='ddmore:environment'])
```

The following information should be used to define environment agents:

Attribute name	Description

prov:label	Human readable description of the environment
version	The underlying operating system
operatingSystem	Version number of the environment
ipAddress	The name of the environment
software	The IP address of the environment (if available)

4.6 Modelling Steps

Existing software packages (such as Pfizer's ePharm, Mango Solutions' "Navigator"[4] and Scinteco's "Improve"[5]) have a concept of a "modelling step".

This is an abstraction that encapsulates the concept of "running a model" (whether it is a parameter estimation, simulation, or some other analysis of a model). This typically includes:

- Datasets consumed by the model
- The model file
- The results generated by running the model with the target software (for example NONMEM or Monolix)

In Prov-O terms, this is equivalent to an Activity of type "estimate" or "simulate".

- The "inputs" to the Step are the files "used"
- The outputs from the Step are the files "generated"
- The software agent is the software used to perform the estimation or simulation
- The user agent is responsible for running the model
- The "plan" is any shell/wrapper script that was used to invoke the target software (e.g. psn.execute)

[4] http://www.mango-solutions.com/wp/products-services/products/navigator/
[5] http://www.scinteco.com/

5 User actions to capture

5.1 Principles

The mechanism by which we capture the actions taken by the user is based on the following principles:

1. All “file type” entities referenced within the workflow database must be persistent. This means that is must be stored within a reliable, secure, referenceable and perpetual system - for instance a version control system.
2. All records stored within the database are **immutable** - i.e. they cannot be changed once they are stored. If a change it necessary, then it should be captured as a new record, which replaces the previous one.
3. All actions that create new entities must also be captured as an Activity, to keep a record of who performed that action and when.
4. Activity IDs can be created on demand by any client, as long as they retain the namespace that indicates which project the activity belongs to, and they are guaranteed to be unique
5. The ID of Entities that refer to a file in the version control system should be comprised of the Repository URL, the ID of the commit that includes this change, and the location of the file within that commit.
6. The ID of Entities that do *not* refer to files in a version control system can be created on demand by the client, as long as they retain the namespace, and they are guaranteed to be unique.

5.2 Scenarios

The following scenarios summarize the actions that a user may perform on entities, and indicate how they will be captured in PROV-N.

For brevity, the start/end document tags, and the namespaces are omitted in all but the first example.

5.2.1 A “message” is sent to the Thoughtflow server

This example captures any scenario where a message is sent to the server. This could be because a file has been committed to the repository, or some other information will be added to the server (for example – an Entity has been ascribed a Description).

All interactions with the model repository happen with a “Bundle”, which allows us to attach “provenance” to a set of changes. This is also necessary to allow entities to be reference in later message. The examples in this section build on this concept, by only defining the content of the Bundle. The information to capture is:

- There was an Activity, for example “commit changes to the repository”, along with a date and time
- The user agent that was responsible for the message

- The contents of the message (in the bundle)

An example of such a document (in PROV-N) is as follows:

```
document
  default <http://www.ddmore.eu/#>
  prefix xsd <http://www.w3.org/2001/XMLSchema#>
  prefix ddmore <http://www.ddmore.eu/#>
  prefix prov <http://www.w3.org/ns/prov#>
  prefix repo <https://github.com/msmith/MDLProject#>
  prefix vcs <https://github.com/#>

  agent(msmith)

  entity(repo:abc123, [prov:type='prov:Bundle',
  ddmore:type="ddmore:commit",
  vcs:repo="https://github.com/msmith/MDLProject",
  vcs:type="git", vcs:commitId='abc123', vcs:branch='master',
  prov:label="Initialised project data",
  vcs:message="Initialised project data"])

  activity(repo:123456, 2016-07-20T16:02:36Z, -,
  [prov:type="ddmore:commit"])
  wasAssociatedWith(repo:123456, msmith, -)
  wasGeneratedBy(repo:abc123, repo:123456, -)

  bundle repo:abc123
    entity(repo:abc123/data/warfarin_conc.csv,
    [prov:label="warfarin_conc.csv",
    prov:location="data/warfarin_conc.csv",
    prov:type="ddmore:dataset"])
  endBundle

endDocument
```

In this case:

- There is a bundle in this message,
  - It is a commit
    - to a Git repository
    - with a commit ID of `abc123`
    - onto the master branch
    - with the commit message “Initialised project data”
- User `msmith` was responsible for this bundle (commit)
- The bundle is linked to a “`ddmore:commit`” Activity

- The bundle itself was created as a result of this commit
- The content of the bundle is the file "`warfarin_conc.csv`" which is located in the folder "`data`"

5.2.1.1 Referring to information in other bundles

Every provenance document should be standalone and internally consistent. PROV-O supports cross referencing entities across bundles with the "mentionOf" statement.

A ***mention*** ◊ relation, written `prov:mentionOf(local, remote, bundle)` in PROV-N, has:

- *specificEntity*: an identifier (`local`) of the entity that is a mention of the general entity (`remote`);
- *generalEntity*: an identifier (`remote`) for an entity that is described in bundle `bundle`.
- *bundle*: an identifier (`bundle`) of a bundle that contains a description of `remote` and further constitutes one additional aspect presented by `local`

In practice, when an entity was created in another bundle, a "mention of" statement is required if a link is being added between the previous entity and the entity in the latest bundle.

5.2.2 Create child model

Model P exists within a project. It was committed in a previous bundle (`abc123`).

Model Q is a new file that is a child model of Model P. It is being committed in bundle `def456`, by user `jwilkins`.

```
mentionOf(temp, repo:abc123/modelP, repo:abc123)

entity(repo:def456/modelQ, [ prov:type="ddmore:model",
prov:location="/models/modelQ.ctl"] )

activity(repo:cloneModel, -, -, [prov:type="ddmore:clone"])
wasAssociatedWith(repo:cloneModel, jwilkins)
used(repo:cloneModel, temp)
wasGeneratedBy(repo:def456/modelQ, repo:cloneModel)

wasDerivedFrom(repo:modelQ, temp, repo:cloneModel,
[prov:type="ddmore:specialization"] )
```

5.2.3 Weak links between models

Model P exists within a project. It was committed in a previous bundle (`abc123`).

Model P has influenced the development of Model Q, but model Q is not a child.

```
mentionOf(temp, repo:abc123/modelP, repo:abc123)
entity(repo:modelQ, [ prov:type="ddmore:model",
prov:location="/models/modelQ.ctl”] )
wasInfluencedBy(repo:modelQ, temp )
```

The “wasInfluencedBy” relationship can be augmented with additional descriptions to ascribe extra meaning to it:

```
wasInfluencedBy(repo:modelQ, temp,
 [prov:type=”sharesCovariateModel”]] )

wasInfluencedBy(repo:modelQ, temp,
 [prov:type=”sharesStructuralModel”]] )
```

These descriptions can be added on demand but should be consistent within an organisation to allow this information to be queried and extracted from the database.

5.2.4 Updating a file

User A alters the content of a file.

Version n of the file is no longer available in the latest version of the “project”.

As the previous file is not there any longer, it should be captured as a revision, version n+1.

This new version is committed in bundle `def456`

```
mentionOf(temp, repo:abc123/modelA, repo:abc123)

entity(repo:def456/modelA, [ prov:type="ddmore:model",
prov:location="/folder/modelA.ctl”] )
activity(repo:updateModel)
wasGeneratedBy(repo:def456/modelA, repo:updateModel)
used(repo:updateModel, temp)
wasAssociatedWith(repo:updateModel, userA)
wasDerivedFrom(repo:def456/modelA, temp, repo:updateModel,
[prov:type='prov:Revision'])
```

See the solution design for more details on entity ids.

5.2.5 Moving a file

A user moves a file within a project into a different location in commit `def456`

The file is otherwise unchanged. The file is no longer available at the previous location.

As the previous file is not there any longer, it should be captured as a revision.

```
mentionOf(temp, repo:abc123/modelA, repo:abc123)
entity(repo:def456/modelA, [ prov:type="ddmore:model",
prov:location="/new/folder/modelA.ctl”] )

activity(repo:moveEntity)
wasGeneratedBy(repo:def456/modelA, repo:moveEntity)
used(repo:moveEntity, temp)
wasAssociatedWith(repo:moveEntity, jwilkins)

wasDerivedFrom(repo:def456/modelA, temp, repo:moveEntity,
[prov:type='prov:Revision'])
```

5.2.6 Adding metadata to an Entity

In this case, the user wishes to add some description to the Entity, for instance:

- update the label (“final”, “base”, <none>)
- update the QC status (“true”, “false”)
- update its pivotal status (“true”, “false”)

According to principle 2, the record of an Entity should be immutable in the workflow database. Moreover, when a description is added to an Entity, the entity has not changed – in terms of provenance, there is no new revision of the entity to track. What has changed is the entity description, not the entity it is describing.

We have two independent entities, the entity, and the entity description and relate them with the wasInfluencedBy relationship.

Say model A was committed in Bundle abc123

User jwilkins ascribes a description to it.

The description Entity is not attached to a specific commit, as there is no concrete file that backs it up in the version control system.

However, the description Entity was included in a Bundle. In the example in 5.2.1, the Bundle was a commit to a version control system. This time, there is no Commit, so the type of Bundle is different.

```
document
  default <http://www.ddmore.eu/#>
  prefix ddmore <http://www.ddmore.eu/#>
  prefix prov <http://www.w3.org/ns/prov#>
  prefix repo <https://github.com/msmith/MDLProject#>

  agent(jwilkins)
```

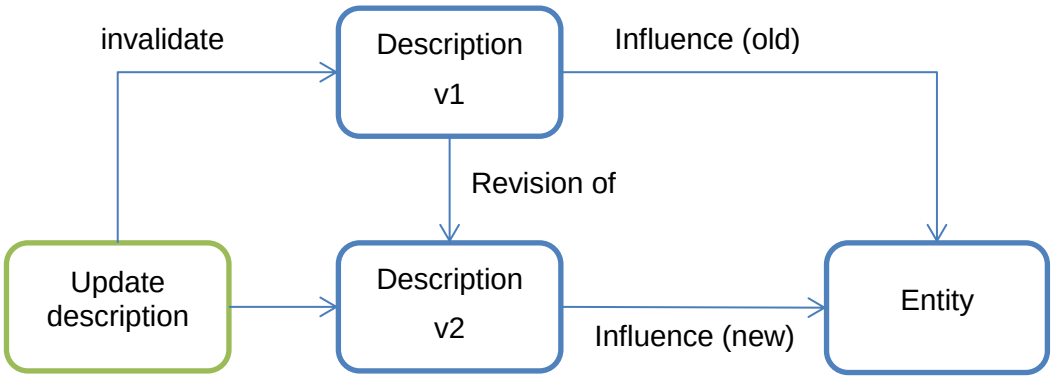
```
  entity(repo:789bca, [prov:type='prov:Bundle',
  ddmore:type='ddmore:description'])
  activity(repo:789789, 2016-07-20T16:02:36Z, -,
  [prov:type="ddmore:description"])
  wasAssociatedWith(repo:789789, jwilkins, -)
  wasGeneratedBy(repo:789bca, repo:789789, -)

  bundle repo:789bca
    mentionOf(temp, repo:abc123/modelA, repo:abc123)
    entity(repo:789bca/description,
    [prov:label="ModelA description",
    prov:type="ddmore:description", ddmore:pivotal="true" ])
    wasInfluencedBy(temp, repo:789bca/description,
    [prov:type="ddmore:describes"])
  endBundle

endDocument
```

All the "description" fields will be attached to the attributes on the Entity. The following table lists the field names and their possible values.

Metadata field name	Meaning	Possible values
ddmore:qcStatus	Whether this entity has passed or failed a QC process	<blank> (indicates it has not been QCed) true (passed QC) false (failed QC)
ddmore:final	This is the final model	true / false
ddmore:base	This is the base model	true / false
ddmore:pivotal	This model is pivotal	true / false

5.2.7 Updating the description of an Entity

The following diagram shows the graph that represents a description being updated (green is activity, blue is an entity)



This is achieved as follows (building on the previous message):

```
document
  default <http://www.ddmore.eu/#>
  prefix ddmore <http://www.ddmore.eu/#>
  prefix prov <http://www.w3.org/ns/prov#>
  prefix repo <https://github.com/msmith/MDLProject#>

  agent(msmith)
  entity(repo:111aaa, [prov:type='prov:Bundle',
  ddmore:type='ddmore:description'])
  activity(repo:4444, 2016-07-20T16:02:36Z, -,
  [prov:type="ddmore:description"])
  wasAssociatedWith(repo:4444, msmith, - )
  wasGeneratedBy(repo:111aaa, repo:4444, - )

  bundle repo:111aaa
    mentionOf(oldDesc, repo:789bca/description, repo:789bca)
    wasInvalidatedBy(oldDesc, repo:4444, 2016-07-20T16:02:36Z)

    mentionOf(modelRef, repo:abc123/modelA, repo:abc123)

    entity(repo:111aaa/description,
    [prov:label="ModelA description",
    prov:type="ddmore:description", ddmore:pivotal="false" ])

    wasDerivedFrom(repo:111aaa/description, oldDesc,
    repo:4444, [prov:type='prov:Revision'])

    wasInfluencedBy(modelRef, repo:111aaa/description,
    [prov:type="ddmore:describes"])
```

```
  endBundle
endDocument
```

5.2.8 QC Status

Section 4.2.2 introduced the entity type “ddmore:qc”. This type of Entity behaves in the same was as a description Entity, but only captures the QC status of the Entity.

This allows the QC status to evolve separately from the model description.

While attaching a QC status to an Entity is similar to the scenario above, the QC activity is a different activity type.

There may also be Entities created as a result of the QC Activity. The actual QC activity itself, therefore, will generate multiple bundles to capture:

- The files being committed to the version control system
- The updated QC description entity

The following document shows how a QC Activity should be recorded.

```
document
  default <http://www.ddmore.eu/#>
  prefix ddmore <http://www.ddmore.eu/#>
  prefix prov <http://www.w3.org/ns/prov#>
  prefix repo <https://github.com/msmith/MDLProject#>

  agent(pchan)
  activity(repo:222222, 2016-07-20T16:02:36Z, -,
  [prov:type="ddmore:qc"])
  wasAssociatedWith(repo:222222, pchan, -)

  entity(repo:444444, [prov:type='prov:Bundle',
  ddmore:type='ddmore:description'])
  wasGeneratedBy(repo:444444, repo:222222, -)

  entity(repo:555555 [prov:type='prov:Bundle',
  ddmore:type="ddmore:commit",
  vcs:repo="https://github.com/msmith/MDLProject",
  vcs:type="git", vcs:commitId='555555', vcs:branch='master',
  prov:label="Added QC report", vcs:message="Added QC
  report"])
  wasGeneratedBy(repo:555555, repo:222222, -)

  mentionOf(modelRef, repo:abc123/modelA, repo:abc123)

  bundle repo:444444
    entity(repo:444444/description,
```

```
    [prov:type="ddmore:qc", ddmore:qcStatus="true" ])
    wasInfluencedBy(modelRef, repo:444444/description,
    [prov:type="ddmore:describes"])
  endBundle

  bundle repo:555555
    entity(repo:555555/QCReports/Report.doc,
    [prov:type="ddmore:document" ])
  endBundle

endDocument
```

An Activity can generate as many outputs as required - so there could be multiple QC descriptions, reports, or decisions.

5.2.9 Making an Assumption/Decision

The pattern for making an assumption or decision is identical in concept to any other Activity that generates an output, as the Assumption/Decision is backed up by a file within the version control system. The message would be comprised of the following:

- The definition of the bundle (i.e. the commit to the version control system, with the user responsible and the commit message)
- The contents of the bundle:
  - The Assumption/Decision entity that points to the XML
  - The Activity with type "ddmore:assumption"
  - The connection between the "activity" and the assumption being created ("wasGeneratedBy")

An example of the bundle contents is shown below:

```
entity( repo:87654/assumptions/assumption1.xml,
[prov:type="ddmore:assumption",
mid3:assumptionType="pharmacological",
mid3:established="new", mid3:testable="true" ] )
activity(repo:374786, 2016-07-20T16:02:36Z, -,
[prov:type="ddmore:assumption"])
wasGeneratedBy(repo:87654/assumptions/assumption1.xml,
repo:374786)
wasAssociatedWith(repo:374786, msmith, - )
```

5.2.10 Updating Entities revisited

Consider the situation where an Entity E has a description D and a QC status Q.

The Entity is revised, in activity A, creating a revision of this Entity, with new ID E1.

The “commit” activity:

- invalidates the QC status entity “Q”
- Creates a new `wasInfluencedBy` record between the Description entity D and the new Entity E1.

If the Description Entity then changes with Activity A2, creating description entity D1

- Entity D is invalidated
- A new `wasInfluencedBy` record is created between Entity E1 and description entity D1
- However, no new `wasInfluencedBy` records are created between the original entity E and the new description D1.

It is the responsibility of the Thoughtflow server to maintain this when Entities are updated.

5.2.11 Template models

Model A exists within a central repository. It was committed in the bundle with commit ID `edc456`

Model B is a new file that is added into the project, by directly copying model A.

```
document
  default <http://www.ddmore.eu#>
  central <http://wwwdev.ebi.ac.uk/biomodels/model-
  repository#>
  prefix repo <https://github.com/msmith/MDLProject#>

  entity(repo:88888, [prov:type='prov:Bundle',
  ddmore:type=”ddmore:commit”,
  vcs:repo="https://github.com/msmith/MDLProject",
  vcs:type="git", vcs:commitId='888888', vcs:branch='master',
  prov:label="Imported model", vcs:message="Imported model"])

  activity(repo:123456, 2016-07-20T16:02:36Z, -,
  [prov:type="ddmore:commit"])
  wasGeneratedBy(repo:888888, repo:123456, -)

  bundle repo:888888
    mentionOf(remoteModel, central:edc456/modelA,
    central:edc456)

    entity(repo:modelB, [ prov:type="ddmore:model",
    prov:location="/models/modelB.ctl”] )
```

```
    activity(copyIntoProject, -, -,
    [prov:type=”ddmore:import”])

    wasDerivedFrom(repo:modelB, remoteModel, copyIntoProject,
    -, -, [prov:type='prov:PrimarySource' ])

  endBundle
endDocument
```

6 Querying the Workflow Store

6.1 Storing information in the Workflow store

The workflow store will handle provenance documents sent in the PROV-JSON format.

6.2 Obtaining the model development tree

The purpose of this query is to get an overview of the model development tree

When this is returned, a client will be able to display a tree that represents how the model(s) in a project have evolved during the lifetime of the project, from the base to the final model.

The query will:

- Get all the entities of prov:type “ddmore:model”
- Get all the derivedFrom relationships between those models, where prov:type = “ddmore:specialisation”
- Make sure that all Entities returned are the “latest” revision

The query should also return any Description Entities that have “Influence” over the Models, so that the client can also display corresponding descriptive information.

6.3 Getting Entities

The purpose of this query is to find out information about entities in a project, specifically:

- Following relationships between an Entity and other Entities, such as what entity is derived from / influenced by / a revision of a given entity
- Getting entities of a specific type (e.g. Decisions or Assumptions)

The query will look for

- entity -> derived from -> ? (and/or)
- entity -> revision of -> ?

This query should also work in the opposite direction, as these relationships can be bi-directional.

6.4 Getting Activities

The purpose of this query is to finding information about actions taken within a project, specifically:

- What activity generated an entity
- What activity used an entity
- Get information about a specific activity
- Locate activities of a specific type, e.g. qc, estimate.

The request contains an activity id or an entity id, a repository, some relationships to follow, and the number of transitive relationships to follow

If you provide an entity it looks for activities that are connected to it via the given relationship(s)

If you provide an activity id it looks up that specific activity

Initially, the query will look for

- entity -> used -> ? (and/or)
- ? -> generated -> entity

And return those activities and entities

7 Solution Design

7.1 Provenance Concepts

7.1.1 Identifiers

Every entity, activity and agent within a workflow repository must have a unique identifier.

This identifier must also be globally unique.

It is not important that this identifier is human readable; prov:label can be used to provide some human readable text/description.

7.1.2 Entity IDs

In order to maximise interoperability across the different components in the system, the entity identifiers should also be *predictable*, so it is possible to reverse engineer identifiers if necessary from other information, and locate an entity from its identifier.

An identifier will be made up of the combination of:

- The URL of the repository, encoded in the namespace
- The ID of the “commit” created when the entity inserted into the version control system
- The location of the entity from the root of the version control system

Examples of this are:

```
document
  default <http://www.ddmore.eu/>
  repo < http://git.mango.local/ddmore/analysis_project>

  entity(repo:9d2e4cd9af541a56942aac15272d2a82f7062357/folder/
  modelA.ctl, [ prov:type="ddmore:model",
  prov:location="/folder/modelA.ctl”] ))

endDocument


document
  default <http://www.ddmore.eu/>
  repo <svn://demo2.mango-
  solutions.com/opt/mango/modspace/svn/modspace>

  entity(repo:538 /trunk/5557/Semiphysiological artemisinin
  PK/Models/Executable_semiphysiological_artemisinin_pk.mod, [
  prov:type="ddmore:model", prov:location="
  /trunk/5557/Semiphysiological artemisinin
  PK/Models/Executable_semiphysiological_artemisinin_pk.mod”]
  ))

endDocument
```

For the purposes of readability, this document does not use this format in examples. Appendix A includes examples of “correct” documents.

7.1.3 Activity IDs

Activity IDs can be created by clients using their own local tools, as long as they are Globally Unique. For instance, if the client is running Java, it could use `java.util.UUID` class. If the client is running on the browser, it could use a function that uses the Math.random utility.

The Activity ID must have the namespace of the project in which the action has taken place.

7.1.4 Bundles

A bundle is a named set of provenance descriptions, and is itself an Entity, so allowing provenance of provenance to be expressed.

We will use a bundle to encapsulate a set of provenance statements that occurred at the same time, as a type of transaction.

There are three situations when a bundle will be used to collect together these statements:

1. As a single “commit” to the version control system (containing file updates, deletions, additions and movements)
2. On the completion of an activity
3. When the user updates an Entity description/QC status

7.2 Components

7.2.1 Version Control System

All entities that are tracked in the workflow datastore must be stored within a version control system.

Version control systems typically have the concept of a “repository” that is used to indicate where the entities are stored. In both SVN and Git this represented as a URI that can be used to uniquely locate that repository within an organisation.

For example:

https://github.com/pharmml/lib-metadata

https://sourceforge.net/p/ddmore/thoughtflow-store-server/ci/master/tree/

As all entities should be stored in a repository, and a URL is used to uniquely identify that repository, the “namespace” can be used to link an entity to a storage location.

A sample document would be:

```
document
  default <http://www.ddmore.eu/>
  repo < http://git.mango.local/ddmore/analysis_project>

  entity(repo:abc-123, [ prov:type="ddmore:model",
  prov:location="/folder/modelA.ctl”] ))

endDocument
```

This document specifies that the entity with id “abc-123” is located in the folder/file “/folder/modelA.ctl” within the repository “http://git.mango.local/ddmore/analysis_project”

7.2.1.1 VCS Hooks

The Version Control System will have “hooks” applied to it so that when there is a commit to it, then a message is broadcast indicating:

- The files added/updated/deleted in the commit
- Who made the change
- The commit message

This message is picked up by a VCS monitor and translated into a provenance document that encodes:

- New entities within the repository
- Revisions to existing files
- Invalidation of deleted files

7.2.2 Provenance Infrastructure

The Provenance Infrastructure is comprised of a number of discrete micro services with clear and narrow responsibilities for handling specific messages as they are propagated throughout the system. The microservices are hosted in an Apache Kafka messaging system, and managed via Apache Zookeeper.

The code for the Provenance Infrastructure is stored on Sourceforge at ()

There are:

7.2.2.1 Hook monitor

Codenamed Renoir.

Receives WebHook events, currently from Git repositories and publishes these events onto the webhook-event topic.

7.2.2.2 Hook translator

Codenamed Gladys.

Translates WebHook VCS events into concrete VCS events and posts them onto the vcs-event topic.

7.2.2.3 VCS Translator

Codenamed Potter.

Converts events into provenance documents through configured templates and posts them onto the prov-payload topic.

7.2.2.4 PROV forwarder

Codenamed Prudence

Receives provenance documents and uploads them to a Thoughtflow server capable of processing provenance documents.

7.2.2.5 Activity Monitor

Codenamed Zita

Spring Boot / Spring Integration application for receiving partial providence information and storing it until it needs to be assembled into a full providence document.

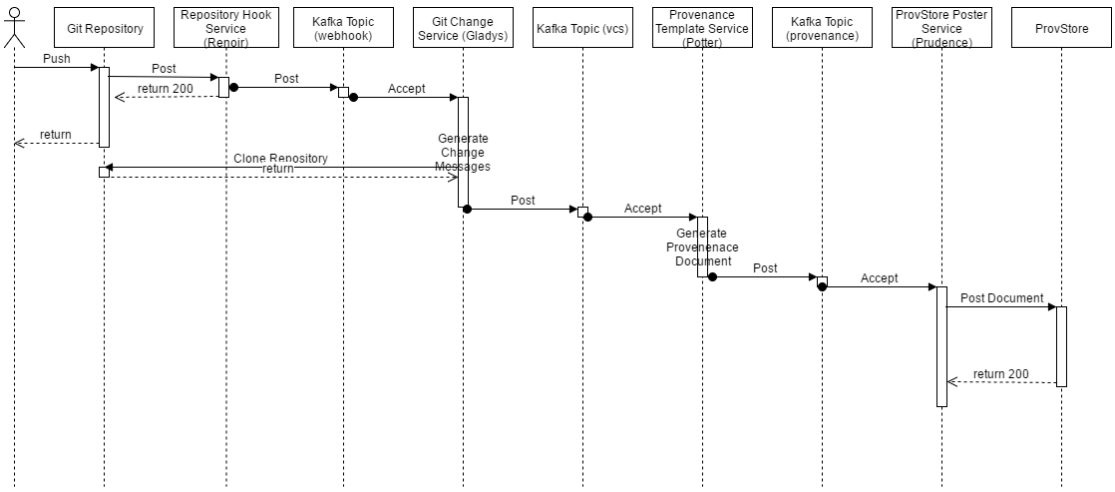
The application receives the notification on the `provenance` topic and processes the notification. The default port for the web management interface is `10060`.

This is used to track long running activities such as parameter estimations or simulations. Zita intercepts the initial request a keeps a record of the Activity starting, along with the initial inputs (the “used”s). When the job completes, the results are collected and the Activity record is completed with the end time.

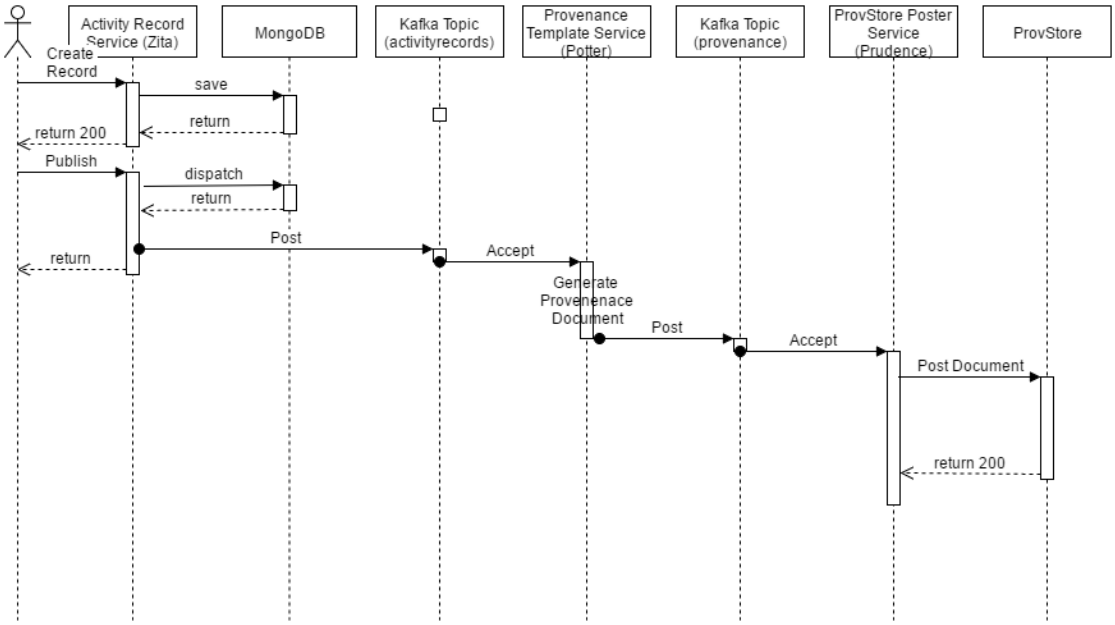
Finally, when the results are committed to the version control system, Zita detects this and creates the appropriate “generated” messages, for forwarding to the Thoughtflow Server.

7.2.2.6 Infrastructure Interactions

The following diagram shows how these components interact when a file is committed to the version control system:



The following diagram shows the process of an Activity being started and monitored by Zita. Note that this shows Prudence posting the Document onto ProvStore (The University of Southampton's publically available Provenance Store) but can be configured to push information to the Thoughtflow Server instead.



7.2.3 Thoughtflow Repository

The University of Southampton's ProvStore is implemented using a traditional relational database. The DDMoRe Thoughtflow repository will be implemented using a Graph database, so we can support the following sorts of inferences:

- When/Should an Entity is change, what are the complete downstream impacts of this change? (i.e. every activity where this entity was used, and the corresponding entities that were generated by that activity, right the way through the tool chain)
- Follow back the complete chain that generated a particular entity (right the way back to source data)
- What should be re-run to "restore" an Entity that is now out of date.

There are several property graph databases on the market such as Neo4J and OrientDB, but we have chosen to use an RDF graph database to store our provenance information as sequences of RDF triples. This is more in keeping with the DDMoRe ontology knowledge base server, which also uses RDF to store information that describes the models themselves (such as the therapeutic area and type of model). In future it is envisaged that these schemas can be combined to ask sophisticated questions such as "locate models that are a final model and have therapeutic area "diabetes""

We will utilise the University of Southampton's ProvToolbox (https://github.com/lucmoreau/ProvToolbox) for processing Provenance Documents and translating them between different formats (PROV-JSON, RDF)

The repository will be part of the Thoughtflow Server, which will be accessed via a REST API for storing and retrieving Provenance Documents. The REST API will support the queries described in section 6.

Documents that created provenance records are submitted in PROV-JSON format.

Queries are invoked as POST requests with a JSON payload.

See the JSON schemas in the Thoughtflow Server codebase hosted on Sourceforge at (https://sourceforge.net/p/ddmore/thoughtflow-store-server/ci/master/tree/) for formal definitions of the messages.

7.2.4 R package

The DDMoRe R package provides tools for reading in, manipulating and executing models written in MDL. When a model is run, it is submitted to the Task Execution Service which routes the request, via the framework, to the location where the model will be run.

An additional R package will be developed that can be used from the MDL-IDE or any other location that supports user driven actions that cannot be implicitly generated by the provenance infrastructure, such as:

- Cloning a model
- Making an assumption
- Making a decision

The R package, in response to a function such as "clone(model)", will:

- Create a copy the model
- Commit the copy to the version control system
- Send the message that the clone was derived from the source model to the Thoughtflow server

The R package will also support running queries against the ThoughtFlow server to read and visualise the ThoughtFlow graph, as the model development tree, and as the task tree, which will include activities, and their inputs and outputs.

7.2.5 Task Execution Service

The Task Execution service acts as a middleman, routing requests to run a task, through to the software application that will be responsible for executing the task.

When the Task Execution service receives a request, it captures:

- The start time of the activity
- The inputs to the request
- The software to be executed
- The agent (user) involved

When the task completes, the outputs are captured, along with the end time. It then generates an Activity message, with the entities involved, and forwards it to the Thoughtflow server.