# EMMAA Documentation

*Release 1.15.0*

**EMMAA developers**

**May 30, 2021**

# Contents


**1 EMMAA Architecture and Approach** **1**
- 1.1 Model Assembly and Updates . . . 2
  - 1.1.1 Cancer types of interest . . . 2
  - 1.1.2 Model availability . . . 3
  - 1.1.3 Defining model scope . . . 3
  - 1.1.4 Deriving relevant terms for a given type of cancer . . . 3
  - 1.1.5 Updating the network . . . 4
  - 1.1.6 Machine-reading . . . 4
  - 1.1.7 Automated incremental assembly . . . 4
- 1.2 Meta-Model . . . 5
  - 1.2.1 Initial specification of annotation guidelines . . . 6
  - 1.2.2 EMMAA currently supports "does X…" queries for PySB models . . . 6
  - 1.2.3 Annotations required for "what if" queries . . . 7
  - 1.2.4 Annotations required for open-ended "relevance" queries . . . 7
- 1.3 Model Testing and Analysis . . . 7
  - 1.3.1 Model test cycle deployed on AWS . . . 8
  - 1.3.2 Test conditions generated automatically . . . 8
  - 1.3.3 General EMMAA model testing framework . . . 9
  - 1.3.4 Model queries from users . . . 10
  - 1.3.5 Pre-registered queries and notifications . . . 10
- 1.4 Model Analysis Query Language . . . 11
  - 1.4.1 Structural properties with constraints . . . 11
  - 1.4.2 Path properties with constraints . . . 13
  - 1.4.3 Simple intervention properties . . . 14
  - 1.4.4 Comparative intervention properties . . . 14

**2 EMMAA Dashboard** **17**
- 2.1 EMMAA Models Page . . . 18
  - 2.1.1 Link to statement details . . . 18
  - 2.1.2 Model Tab . . . 18
  - 2.1.3 Tests Tab . . . 18
  - 2.1.4 Papers Tab . . . 20
  - 2.1.5 Curation Tab . . . 20
  - 2.1.6 Load Previous State of Model . . . 26
- 2.2 EMMAA Statement Evidence Page . . . 26
- 2.3 EMMAA All Statements Page . . . 28

2.4 EMMAA Individual Paper Page . . . 28
2.5 EMMAA Model Queries . . . 29
2.5.1 Static Queries . . . 29
2.5.2 Dynamical Queries . . . 30
2.5.3 Open Search Queries . . . 30
2.5.4 Waiting for results . . . 32
2.5.5 Logging In and Registering a User . . . 33
2.5.6 Subscribing to a Query . . . 33
2.5.7 Email Notifications of Subscribed Queries . . . 34
2.6 Test and query result interpretation . . . 34
2.7 EMMAA Detailed Test Results . . . 37
2.7.1 Results for Different Model Types . . . 37
2.7.2 Non-passing Tests . . . 38

**3 EMMAA modules reference** **39**
3.1 EMMAA Statement (`emmaa.statements`) . . . 39
3.2 EMMAA Model (`emmaa.model`) . . . 40
3.3 EMMAA Model Test Framework (`emmaa.model_tests`) . . . 45
3.4 Analyze model test results (`emmaa.analyze_tests_results`) . . . 49
3.5 Query classes (`emmaa.queries`) . . . 53
3.6 Process model queries (`emmaa.answer_queries`) . . . 55
3.7 Priors (`emmaa.priors`) . . . 55
3.7.1 Literature Prior (`emmaa.priors.literature_prior`) . . . 56
3.7.2 TCGA Cancer Prior (`emmaa.priors.cancer_prior`) . . . 57
3.7.3 Gene List Prior (`emmaa.priors.gene_list_prior`) . . . 57
3.7.4 Reactome Prior (`emmaa.priors.reactome_prior`) . . . 58
3.7.5 Querying Prior Statements (`emmaa.priors.prior_stmts`) . . . 59
3.8 Readers (`emmaa.readers`) . . . 59
3.8.1 AWS reader (`emmaa.readers.aws_reader`) . . . 59
3.8.2 INDRA DB client reader (`emmaa.readers.db_client_reader`) . . . 60
3.9 EMMAA's Database (`emmaa.db`) . . . 60
3.9.1 The Database Schema (`emmaa.db.schema`) . . . 60
3.9.2 Database Manager (`emmaa.db.manager`) . . . 62
3.10 AWS model update and testing pipeline (`emmaa.aws_lambda_functions`) . . . 64
3.11 xDD client . . . 69
3.12 EMMAA's Subscription Service (`emmaa.subscription`) . . . 70
3.12.1 Notifications functions (`emmaa.subscription.notifications`) . . . 70
3.12.2 Email Service (`emmaa.subscription.email_service`) . . . 73
3.12.3 Email Utilities (`emmaa.subscription.email_util`) . . . 75
3.13 Utilities (`emmaa.util`) . . . 76
3.14 Functions for node and edge filtering (`emmaa.filter_functions`) . . . 77

**4 Configuring an EMMAA model** **79**
4.1 First level fields of config.json . . . 79
4.2 Model update configuration . . . 81
4.3 Model testing configuration . . . 82
4.4 Model queries configuration . . . 83
4.5 Making tests from model configuration . . . 84

**5 ASKE Reports** **87**
5.1 ASKE Month 5 Milestone Report: Lessons Learned . . . 87
5.1.1 Automated model assembly: the challenge of defining scope and context . . . 87
5.1.2 Automated model analysis: benefits of automated model validation . . . 88
5.1.3 Test-driven modeling . . . 89

5.1.4 Exploiting the bidirectional relationship between models and tests . . . 92
5.2 ASKE Month 6 Milestone Report . . . 92
5.2.1 Making model analysis and model content fully auditable . . . 92
5.2.2 Including new information based on relevance . . . 93
5.2.3 Coarse-grained model checking of EMMAA models with directed graphs . . . 93
5.3 ASKE Month 7 Milestone Report . . . 95
5.3.1 Repositioning EMMAA within the ASKE framework of modeling layers . . . 95
5.3.2 Use cases for the EMMAA system (and ASKE systems in general) . . . 96
5.4 ASKE Month 9 Milestone Report . . . 97
5.4.1 Generalizing EMMAA: a proof-of-principle model of food insecurity . . . 97
5.4.2 Extending model testing and analysis to multiple resolutions . . . 98
5.4.3 Implementing an object model for model analysis queries . . . 98
5.4.4 Detecting changes in analysis results due to model updates . . . 99
5.5 ASKE Month 11 Milestone Report . . . 99
5.5.1 Deployment of multiple-resolution model testing and analysis . . . 99
5.5.2 User-specific query registration and subscription . . . 102
5.5.3 An improved food insecurity model . . . 102
5.6 ASKE Month 13 Milestone Report . . . 104
5.6.1 Related work for the EMMAA system . . . 104
5.6.2 System performance statistics . . . 104
5.7 ASKE Month 15 Milestone Report . . . 108
5.7.1 EMMAA Knowledge assemblies as alternative test corpora . . . 108
5.7.2 Time machine . . . 109
5.7.3 Dynamical model simulation and testing . . . 110
5.7.4 Towards push science: User notifications of newly-discovered query results . . . 111
5.8 ASKE Month 18 Milestone Report . . . 111
5.8.1 Expert curation of models on the EMMAA dashboard . . . 111
5.8.2 Viewing and ranking all statements in a model . . . 111
5.8.3 Email notifications . . . 112
5.8.4 A model of Covid-19 . . . 112
5.8.5 Integration of content from UW xDD system . . . 113
5.8.6 Configurable model assembly pipeline . . . 113

**6 ASKE-E Reports** **115**
6.1 ASKE-E Month 1 Milestone Report . . . 115
6.1.1 Overall goals and use cases for the Bio Platform . . . 115
6.1.2 Integration plan for the Bio Platform . . . 115
6.1.3 Progress during the ASKE-E Hackathon . . . 116
6.1.4 Open Search model queries and notifications . . . 117
6.2 ASKE-E Month 2 Milestone Report . . . 118
6.2.1 Push science: EMMAA models tweet new discoveries and explanations . . . 118
6.2.2 Improving named entity recognition in text mining integrated with EMMAA models . . . 120
6.2.3 Making model tests and paths available for use by other applications . . . 120
6.3 ASKE-E Month 4 Milestone Report . . . 120
6.3.1 EMMAA Neurofibromatosis Models and NF Hackathon Prize . . . 120
6.3.2 Rapid initialization of EMMAA models from literature for two new diseases . . . 121
6.3.3 Downloading EMMAA models in alternative formats . . . 121
6.4 ASKE-E Month 5 Milestone Report . . . 122
6.4.1 Semantic filters to improve model analysis . . . 122
6.4.2 Model analysis exploiting ontological relationships . . . 122
6.4.3 Improved reading and assembly of protein chains and fragments . . . 123
6.4.4 Bio ontology optimized for visualization . . . 123
6.5 ASKE-E Month 6 Milestone Report . . . 124
6.5.1 Reading and assembly with context-aware organism prioritization . . . 124

6.5.2 Preparing for the stakeholder meeting . . . 124
6.5.3 Reporting curation statistics . . . 125
6.5.4 Reporting paper level statistics . . . 127
6.5.5 Integrating non-textual evidence with EMMAA models . . . 128
6.6 ASKE-E Month 7 Milestone Report . . . 128
6.6.1 Natural language dialogue interaction with EMMAA models . . . 128
6.6.2 Automatically generated text annotations in context . . . 131
6.6.3 Demonstrations at the stakeholder meeting . . . 132
6.6.4 Developing the EMMAA REST API for flexible integration . . . 133
6.7 ASKE-E Month 9 Milestone Report . . . 134
6.7.1 Integrating the COVID-19 Disease Map community model . . . 134
6.7.2 Notifications about general model updates . . . 138
6.7.3 Figures and tables from xDD as non-textual evidence for model statements . . . 138
6.7.4 Integration with the Uncharted UI . . . 139
6.7.5 Semantic separation of model sources for analysis and reporting . . . 139
6.7.6 Assembling and analyzing dynamical models . . . 140
6.7.7 Creating a training corpus for identifying causal precedence in text . . . 140
6.7.8 Knowledge/model curation using BEL annotations . . . 141
6.7.9 Formalizing EMMAA model configuration . . . 141
6.8 ASKE-E Month 10 Milestone Report . . . 141
6.8.1 Dynamical model analysis . . . 141
6.8.2 Improved EMMAA query UI and REST API . . . 146
6.8.3 Network representation learning for EMMAA models . . . 146

**Python Module Index** **155**

**Index** **157**

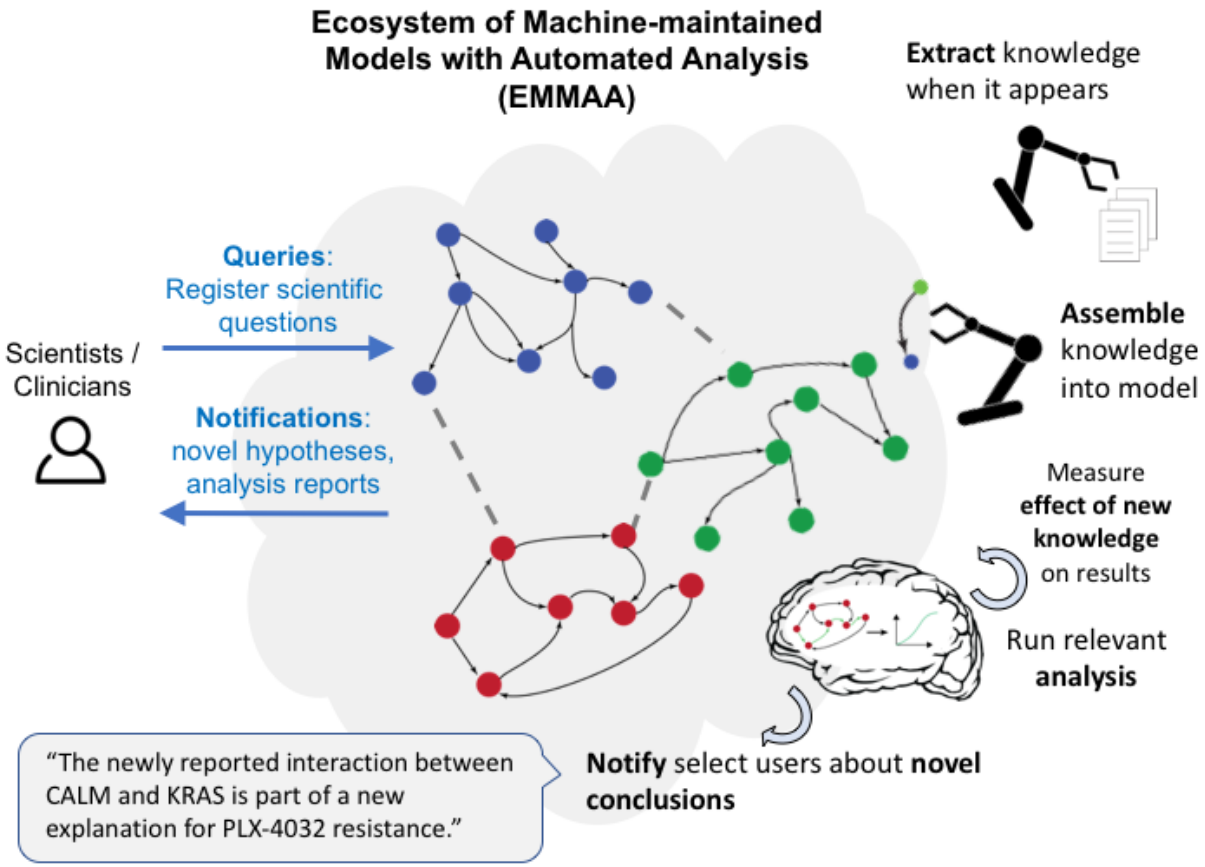
# CHAPTER 1

# EMMAA Architecture and Approach

The Ecosystem of Machine-maintained Models with Automated Analysis is available at http://github.com/indralab/emmaa, with the EMMAA Model Dashboard at http://emmaa.indra.bio.

The main idea behind EMMAA is to create a set of computational models that are kept up-to-date using automated machine reading, knowledge-assembly, and model generation, integrating new discoveries immediately as they become available.

As a key component of the approach, models are automatically tested against experimental observations (also automatically gathered and associated with models). Models are also available for automated analysis in which relevant queries that fall within the scope of each model can be automatically mapped to structural and dynamical analysis procedures on the model. Currently, the Dashboard supports running and registering queries with respect to one or more existing models. In the near future, EMMAA will automatically recognize and report changes to each model that result in meaningful changes to analysis results.




## 1.1 Model Assembly and Updates

### 1.1.1 Cancer types of interest

We start with six cancer types that are particularly relevant due to a combination of frequency of occurrence and lack of adequate treatments. The cancer types we have initially chosen are as follows.

- Acute Myeloid Leukemia (aml)
- Breast Carcinoma (brca)
- Lung Adenocarcinoma (luad)
- Pancreatic Adenocarcinoma (paad)
- Prostate Adenocarcinoma (prad)
- Skin Cutaneous Melanoma (skcm)

Each type is followed by a "code" in parantheses indicating the identifier of the model through which models are organized in the cloud, on AWS S3.

### 1.1.2 Model availability

EMMAA models may be browsed on the EMMAA Dashboard, for more information, see a tutorial to the dashboard here: *EMMAA Dashboard*, and the dashboard itself here: http://emmaa.indra.bio. For example the AML model can be accessed directly at http://emmaa.indra.bio/dashboard/aml.

### 1.1.3 Defining model scope

Each model is initiated with a set of prior entities and mechanisms that take entities as arguments. Search terms to extend each model are derived from the set of entities.

### 1.1.4 Deriving relevant terms for a given type of cancer

Our goal is to identify a set of relevant entities (proteins/genes, families, complexes, small molecules, biological processes and phenotypes) that can be used to acquire information relevant to a given model. This requires three components:

- A method to find entities that are specifically relevant to the given cancer type
- A background knowledge network of interactions between entities
- A method to identify relevant links and entities on the background knowledge network

These methods, as described in the subsections below, are implemented in the *TcgaCancerPrior* (`emmaa.priors.cancer_prior.TcgaCancerPrior`) class.

#### Finding disease genes

To identify genes that are relevant for a given type of cancer, we turn to The Cancer Genome Atlas (TCGA), a cancer patient genomics data set available via the cBio Portal.

We implemented a client to the cBio Portal which is documented here.

Through this client, we first curate a list of patient studies for the given cancer type. These patient studies are tabulated in emmaa/resources/cancer_studies.json.

Next, we query each study with a list of genes (the entire human genome, in batches) to determine which patients have mutations in which genes. From this, we calculate statistics of mutations per gene across the patient population.

#### Finding relevant entities in a knowledge network

Finding relevant entities requires a prior network that can be supplied as a parameter to *TcgaCancerPrior*. We use a network derived from processing and assembling the content of the PathwayCommons, SIGNOR, and BEL Large Corpus databases, as well as machine reading *all* biomedical literature (roughly 32% full text, 68% abstracts) with two machine reading systems: REACH and Sparser. This network has 2.5 million unique mechanisms (each corresponding to an edge).

Starting from the mutated genes described in the previous section, we use a heat diffusion algorithm to find other relevant nodes in the knowledge network. We first normalize the mutation counts by the length of each protein (since larger proteins are statistically more likely to have random mutations which can lack functional significance). We then apply the normalized mutation count as a "heat" on the node in the network corresponding to the gene. When calculating the diffusion of heat from nodes, we take into account the amount of evidence for each edge in the network. The number of independent evidences for the edge (i.e. the number of database entries or extractions from sentences in publications by reading systems) and use a logistic function with midpoint set to 20 by default (parameterizable)

to set a weight on the edge. We use a normalized Laplacian matrix-based heat diffusion algorithm on an undirected version of the network, which can be calculated in a closed form using scipy.sparse.linalg.expm_multiply.

Having calculated the amount of heat on each node, we apply a percentile-based cutoff to retain the nodes with the most heat.

#### Assembling a prior network

Given a set of entities of interest, we turn to the INDRA DB and query for all Statements about these entities. This set of Statements becomes the starting point from which the model begins a process of incremental extension and assembly. This is implemented in `emmaa.priors.prior_stmts`.

### 1.1.5 Updating the network

Given the search terms associated with the model, we use a client to the PubMed web service to search for new literature content.

### 1.1.6 Machine-reading

Given a set of PMIDs, we use our Amazon Web Services (AWS) content acquisition and high-throughput reading pipeline to collect and read publications using the REACH and Sparser systems. We then use INDRA's input processors to extract INDRA Statements from the reader outputs. We also associate metadata with each Statement: the date at which it was created and the search terms which are associated with it. These functionalities are implemented in the `emmaa.readers.aws_reader` module.

As an optimized approach to gathering and reading new publications, we decoupled this step from EMMAA, and it is currently done independently by a scheduled job of the INDRA DB once a day. EMMAA's model update jobs query the DB directly for Statements extracted from the new publications each day, making the model update cycle significantly faster. These queries are implemented in `emmaa.readers.db_client_reader`.

### 1.1.7 Automated incremental assembly

Each time new "raw" Statements are added to the model from new literature results, an assembly process is run which involves the following steps:

- Filter out hypotheses
- Map grounding of entities
- Map sequences of entities
- Filter out Statements with ungrounded entities
- Run preassembly in which exact and partial redundancies are found and resolved
- Calculate belief score for each Statement
- Filter to statements above a configured belief threshold
- Filter out subsumed Statements with respect to partial redundancy graph
- (In some models) filter out Statements representing indirect mechanisms

The set of Statements obtained this way are considered to be "assembled" at the knowledge level. It is this assembled set of Statements that are considered when showing update statistics on the Dashboard. The newly obtained assembled Statements are also evaluated against Statements already existing in the model. Note that The Statements below the

threshold still remain in the "raw" model knowledge and can later advance to be included in the published model if they collect enough evidence to reach the belief threshold.

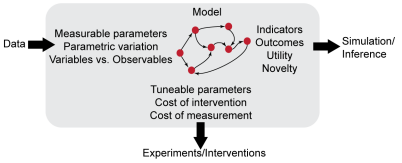
A new Statement can relate to the existing model in the following ways:

- Novel: there is no such mechanism yet in the model
- Redundant / Corroborating: the mechanism represented by the Statement is already in the model, providing new, corroborating evidence for that Statement
- Generalization: the mechanism is a more general form of one already in the model
- Subsumption: the mechanism is a more specific form of one already in the model
- Conflicting: the mechanism conflicts with one already in the model

Currently, the dashboard lists new Statements without explicitly showing what relationship they have to the existing model.

## 1.2 Meta-Model

Analysis of scientific models is typically a manual process in which specific simulation scenarios are formulated in code, executed, and the results evaluated. In EMMAA, models will be semantically annotated with concepts allowing scientific queries to be automatically formulated and executed. The core component of this process will be a *meta-model* for associating the necessary metadata with specific model elements.




As shown in the figure above, the EMMAA meta-model will allow the annotation of:

- relevant entities (e.g., specific genes or biological processes)
- relations/processes (e.g., phosphorylation, activation)
- quantities in model-relevant data (e.g., measured values associated with specific model parameters)
- features of model parameters and observables relevant to subsequent experimental follow-up (e.g.,for example whether a parameter can be experimentally altered or whether measurement of a particular observable is cost-effective)
- higher-level scientific aspects associated with model variables and outcomes, such as the utility associated with particular model states (e.g., decreased cell proliferation)

The EMMAA meta-model allows model elements encoded in different formalisms to be associated with the concepts necessary for automated analysis in EMMAA. For example, a protein initial condition parameter from an executable PySB model could be linked to the EMMAA concepts for a parameter that is *naturally varying, non-perturbable,* and *experimentally measurable.*

While several of these concepts have not been previously implemented in existing ontologies for semantic annotations of biological models, we will aim to reuse terms and concepts from ontologies developed by the COMBINE community where appropriate. These may include:

- MIRIAM (Minimimal Information Required In the Annotation of Models)
- SED-ML (Simulation Experiment Description Markup Language)
- SBO (Systems Biology Ontology)

- KISAO (Kinetic Simulation Algorithm Ontology)
- Biomodels.net qualifiers
- MAMO (the Mathematical Modeling Ontology)
- SBRML (Systems Biology Results Markup Language)
- TEDDY (TErminology for the Description of DYnamics)

### 1.2.1 Initial specification of annotation guidelines

The meta-model will be implemented as a specification that can be implemented in different ways depending on the model type; in this way it will resemble the MIRIAM standard, which is not itself a terminology but rather a set of guidelines for using of *(subject, predicate, object)* triples to link essential model features to semantic concepts.

The EMMAA meta-model establishes several specific concepts and annotation guidelines aimed at automating high-level scientific queries. In particular, the initial specification for model annotation in EMMAA includes the following requirements to support basic simulation and analysis tasks:

1. Model entities (e.g., variables in an ODE model, nodes in a network model) must be linked to identifiers in external ontologies.
2. Entity states (e.g., phosphorylated, mutated, active or inactive proteins) should be identified semantically using an external ontology or controlled vocabulary.
3. Model processes (e.g., reactions in an ODE model, edges in a network model) must be linked to a piece of knowledge including provenance and evidence. In our initial implementation, this will be accomplished using the *has_indra_stmt* relation which will link back to an underlying INDRA statement.
4. Entities participating in processes should be identified with their role (e.g subject or object) for directional analysis.
5. (Optional): if it is not already implicit in the modeling formalism, the model process can be annotated with the *sign* of the process on its participants (i.e., positive or negative regulation).

### 1.2.2 EMMAA currently supports "does X…" queries for PySB models

Annotating a model using the five types of information above supports high-level queries such as: "Does treatment with drug X cause an increase in the phosphorylation of protein Y?" Answering this yes-or-no query makes use of model annotations in the following way:

- Entities in the model representing drug X are identified (#1, above).
- Entities in the model representing phosphorylated Y are identified (#1 and 2).
- Processes with drug X as the subject are identified, as are processes with phosphorylated Y as the object (#4, above).
- The effect of the drug X entities/processes on the phosphorylated protein Y entities/processes are determined using a model-specific analytical procedure, making use of *sign* information if necessary (#5).
- The analysis results are linked back to the knowledge model via *has_indra_stmt* annotations (#3).

We currently have an end-to-end implementation that uses model annotations to answer these types of queries for a single model type: executable dynamical models implemented in PySB. Model annotations are generated as part of the PySB model assembly process in INDRA; for instance see the PySB Assembler code here for an example of how the PySB Annotation class is used to associate entities with their role (subject/object) in a process (#4).

To answer a "does" query like the one specified above, the ModelChecker makes use of these annotations to search for a path through the model's influence map with the appropriate sign.

These types of queries can currently be used to formulate model tests using the *StatementCheckingTest* (`emmaa.model_tests.StatementCheckingTest`), and triggered automatically upon every model update using the testing pipeline described in *Model Testing and Analysis*.

### 1.2.3 Annotations required for "what if" queries

As opposed to a "does X…" query like the example above, which are used to determine the connectivity and sign of causal paths in the model at baseline, a "what if" query indicates a perturbation and involves an open-ended response. For example, consider the following queries:

- "What happens to protein X if I knock out protein Y?"
- "What happens to protein X if I double the amount of drug Y?"
- "What happens to protein X if I decrease its affinity to drug Y?"

Addressing these queries in general requires designating a model *control condition* (e.g., a specific initial state or steady state) that is perturbed by the manipulation of model structure or parameters. This requires the following model features to be identified by additional annotations:

6. Model parameters governing entity amounts
7. Effect of model parameters on the strength of interaction between entities (for example, the forward and reverse rates of a binding interaction both affect the affinity of the interaction, but in opposite ways).

### 1.2.4 Annotations required for open-ended "relevance" queries

Finally, we aim to enable the automation of analysis procedures that are not based on explicit queries but rather aimed at identifying model characteristics that have scientific relevance and value. An example would be to "notify me of mechanistic findings therapeutically relevant to pancreatic cancer." This type of query requires additional annotations on the higher-level biological processes associated with model entities and their scientific relevance. We aim to implement the following additional three annotations for this purpose:

8. Biological processes or phenotypes associated with specific model entities, and their sign (e.g., phosphorylated MAPK1 is positively associated with cell proliferation in pancreatic cancer).
9. A value criterion associated the biological process (e.g., it is therapeutically desirable to *increase* cancer cell apoptosis, and *decrease* cancer cell proliferation).
10. Entity types that represent actionable perturbations. For example, it may be of greater interest to identify a chemical perturbation that yields a desirable affect than a genetic perturbation, because (at least present) chemical perturbations are more experimentally and therapeutically tractable.

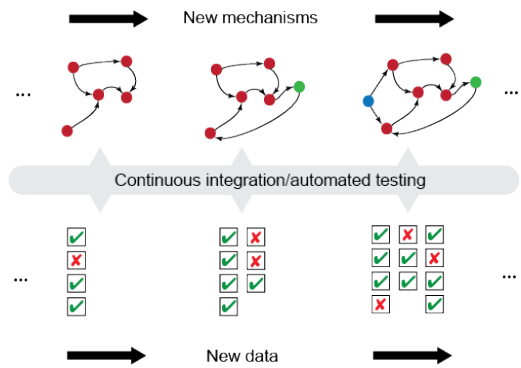
These ten annotation types represent the initial set for the EMMAA cancer models.

## 1.3 Model Testing and Analysis

A key benefit of using semantically annotated models is that it allows models to be automatically validated in a common framework. In addition to automatically extracting and assembling mechanistic models, EMMAA runs a set of tests to determine each model's validity and explanatory scope. We have implemented an approach to model testing that automates

- the collection of test conditions from a pre-existing observational knowledge base,
- deciding which test condition is applicable to which model,
- executing the applicable tests on each model, and

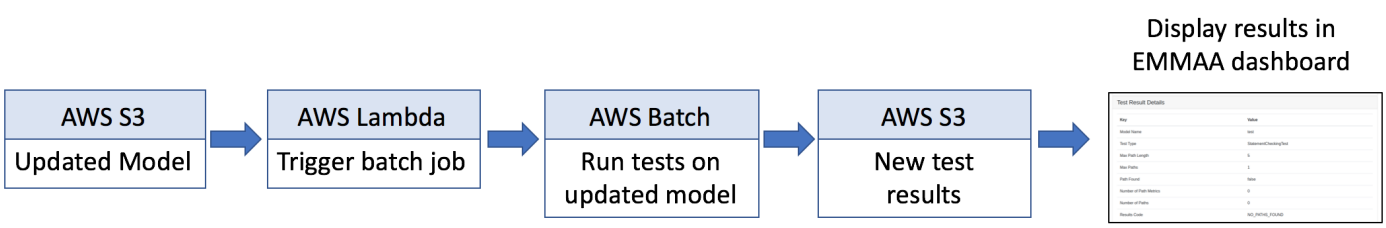
- reporting the summary results of the tests on each model.




The overall concept of automated model testing in EMMAA is shown in this figure. Each time a model is updated with new findings, the model is tested against a set of expected observations or properties. The tests themselves can evolve over time as new observations are collected.

### 1.3.1 Model test cycle deployed on AWS

Whenever there is a change to a model, a pipeline on Amazon Web Services (AWS) is triggered to run a set of applicable model tests. When a model is updated (i.e., with new findings extracted and assembled from novel research publictions), a snapshot of it is deposited on the S3 storage service. A Lambda process monitors changes on S3 and when a change occurs, triggers a Batch job. The Batch job accesses the Dockerized EMMAA codebase and runs the automated test suite on the model. The test results are then deposited on S3. Finally, the new test results are propagated onto the EMMAA Dashboard website. This process is summarized in the figure below.




The code implemented here is available in the following places:

- The Lambda implementation is documented at: `emmaa.aws_lambda_functions`.
- The EMMAA Docker image is available here .

### 1.3.2 Test conditions generated automatically

EMMAA implements a novel approach to collecting observations with respect to which models can be tested. Given a set of INDRA Statements, which can be obtained either from human-curated databases or literature extractions, EMMAA selects ones that represent experimental observations (which relate a perturbation to a potentially indirect downstream readout) from direct physical interaction-like mechanisms. We treat these observational Statements as constraints on mechanistic paths in a model. For instance, the observation "treatment with Vemurafenib leads to decreased phosphorylation of MAPK1", could be satisfied if the model contained a sequence of mechanisms connecting Vemurafenib with the phosphorylation state of MAPK1 such that the aggregate polarity of the path is positive.

As a proof of principle, we created a script which generates such a set of test conditions from the BEL Small Corpus, a corpus of experimental observations and molecular mechanisms extracted by human experts from the scientific

literature. Going forward, we will also rely on observations collected directly from the literature for automated model testing.

The code to generate and run this corpus of test statements is available here.

### 1.3.3 General EMMAA model testing framework

EMMMA contains a test framework in `emmaa.model_tests` with an abtract class interface to connect models with applicable tests and then execute each applicable test with respect to each applicable model. One strength of this abstract class architecture is that it is agnostic to

- the specific content and implementation of each model and test,
- the criteria by which a test is determined to be applicable to a model,
- the procedure by which a test is determined to be satisfied by a model.

It therefore supports a variety of specific realizations of models and tests. The classes providing this interface are the *TestManager* (`emmaa.model_tests.TestManager`), *TestConnector* (`emmaa.model_tests.TestConnector`) and *EmmaaTest* (`emmaa.model_tests.EmmaaTest`).

#### Test conditions mapped to models automatically

EMMAA currently implements a specific set of testing classes that are adequate for our cancer models. This implementation uses the *ScopeTestConnector* (`emmaa.model_tests.ScopeTestConnector`) and *StatementCheckingTest* (`emmaa.model_tests.StatementCheckingTest`) classes in EMMAA. The ScopeTestConnector class uses our meta-model annotations to determine the identity of the concepts in the model as well as in the test, and deems the test to be applicable to the model if all the concepts (i.e. the perturbation and the readout) in the test are also contained in the model.

#### Testing models using static analysis

The StatementCheckingTest class takes a pair of a model and an applicable tests, and determines whether the model satisfies the test as follows. The model is first assembled into a rule-based PySB model object using INDRA's PySB Assembler. The model is then exported into the Kappa framework, which provides static analysis methods, including generating an influence map (a signed, directed graph) over the set of rules in the model. EMMAA then uses INDRA's Model Checker to find paths in this influence map that match the test condition (itself expressed as an INDRA Statement). If one or more such paths are found, the test is assumed to be satisfied, and the results are reported and stored. Otherwise, the model is assumed to to satisfy the test.

An end-to-end model building and testing example is available here.

Going forward, the testing methodology will involve multiple modes of simulation and analysis including also dynamic testing.

#### Human-readable model test reports

A snippet of the test report for a Ras signaling pathway model (see http://emmaa.indra.bio/dashboard/rasmodel) as of 4/1/2019 is shown below, where each "Observation" is expressed in terms of an expectation of model behavior (e.g., "IFG1R phosphorylated on Y1166 activates IRS1") along with a determination of whether the constraint was satisfied (green tick mark if yes, red cross if not), along with a description of the specific way in which the model satisfies the test condition (as human-interpretable English language summary) or the reason for why the model could not satsfy the test condition.

| | | |
|---|---|---|
| EGF bound to EGFR and GRB2 activates RAC1. | ✔ | EGF binds EGFR.<br>EGFR bound to EGF binds EGFR bound to EGF.<br>EGFR bound to EGFR phosphorylates SRC on Y419.<br>Active SRC phosphorylates TIAM1.<br>Active TIAM1 activates RAC1. |
| ELK1 bound to SRF activates ELK1. | ✖ | No path found that satisfies the test statement |
| IGF1R phosphorylated on Y1166 activates IRS1. | ✔ | Active IGF1R phosphorylates IRS1 on tyrosine. |

In a manner analogous to continuous integration for software, EMMAA model testing is automatically triggered on AWS anytime the model or its associated constraints are updated.

### 1.3.4 Model queries from users

Through the EMMAA Dashboard Query page at http://emmaa.indra.bio/query, users can submit specific queries to one or more models simultaneously, that are evaluated immediately by a web service, and the results of the analysis are summarized in a table. For more information, see: *EMMAA Model Queries*.

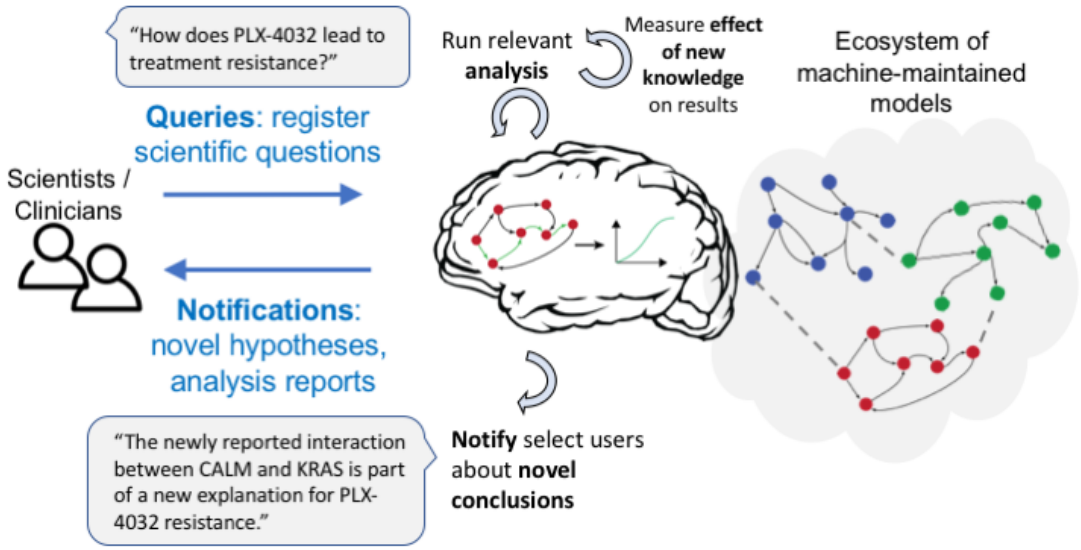
EMMAA currently supports "Path property" queries on its models in a templated form through the Dashboard. However, the types of analysis queries will be extended, and we imagine later supporting natural language-based querying as well. The types of queries EMMAA will support are as follows. We developed a Model Analysis Query Language which specifies these types of properties, see *Model Analysis Query Language*.

- Structural properties with constraints: e.g., "What drugs bind PIK3CA but not PIK3CB?"
- Path properties with constraints: e.g., "How does treatment with PD-325901 lead to EGFR activation?"
- Simple intervention properties: e.g., "What is the effect of Selumatinib on ERK activation by EGF?"
- Comparative intervention properties: e.g., "How is the effect of targeting MEK different from targeting PI3K on the activation of ERK by EGF?"

Each such property maps onto a specific model analysis task that can be run on an EMMAA model, for instance, causal path finding with semantic constraints, or dynamical simulations under differential initial conditions.

### 1.3.5 Pre-registered queries and notifications

Each query can also be "registered" by EMMAA, and evaluated again whenever the model is updated. Currently these registered queries are shared by all users. Going forward, individual users will be able to register their own, personal queries for one or more models of interest. The result of analysis for each property on a given version of the model will be saved. This will then allow comparing any changes to the result of analysis with previous states of the model. If a meaningful change occurs, a notification will be generated to the user who registered the query.




## 1.4 Model Analysis Query Language

This is v1.0 of a specification for a machine-readable description format for the analysis and querying of EMMAA models. The specification uses a JSON format that is easily generated and processed, and is also human-readable and editable.

The specification extends to four, increasingly complex query types:

- Structural properties with constraints
- Path properties with constraints
- Simple intervention properties
- Comparative intervention properties

Note that this specification for defining queries does not explcitly specify the method by which the query is executed, though some query specifications are defined with a certain type of analysis method in mind.

### 1.4.1 Structural properties with constraints

Structural properties of models are evaluated directly at the knowledge-level, in our case at the level of INDRA Statements. Each Statement has a type (Activation, Dephosphorylation, etc.), refers to one or more entities (Agents) as arguments, which themselves can have different types are determined by grounding to an ontology. At an abstract level

Structural property queries can have different "topologies" in terms of the entities they reference including

- unary queries referring to a single Agent alone,
- queries referring to a single Agent and its neighborhood,
- binary queries that refer to two Agents.

Structural property queries may also constrain the type of the Statement and Agent.

### Specifying topology

Structural queries have multiple *subtypes* based on the topology of the query:

- binary_directed: specifies two Agents, a *source* and a *target*, between which, a directed relationship is queried.
- binary_undirected: specifies two Agents in an *agents* list, in arbitrary order, and relationship direction is of interest in the query.
- neighborhood: specifies a single *agent* around which a relationship in any direction (incoming, outgoing, undirected) is of interest.
- to_target: specifies a single Agent as a *target* and only incoming relationships are of interest.
- from_source: specifies a single Agent as a *source* and only outgoing relationships are of interest.
- single_agent: specifies a single *agent* with the query focusing on a property of the Agent itself rather than any relationships.

Each Agent is defined via its name, and optionally, groundings, for more information, see the relevant entry of the INDRA JSON Schema: https://github.com/sorgerlab/indra/blob/master/indra/resources/statements_schema.json#L77

### Entity constraints

Entity constraints (*entity_constraints*) can be added to the query, these can constrain the *type* (protein, chemical, biological process, etc.) and *subtype* (kinase, transcription factor, etc.) of the Agents of interest.

### Relationship constraints

Relationship constraints can be specified by describing the *type* of Statement establishing the relatonship.

### Examples

Example: "What kinases does BRAF phosphorylate?"

```
{"type": "structural_property",
 "subtype": "from_source",
 "source": {
  "type": "agent",
  "name": "BRAF"
  },
 "entity_constraints": [
  {"type": "protein",
   "subtype": "kinase"}
  ],
 "relationship_constraints": [
  {"type": "Phosphorylation"}
 ]
}
```

### Ideas for extension

The constraints could be generalized to allow logical formulae over entity types and relations.

### 1.4.2 Path properties with constraints

Path properites of models are evaluated at a lower level than simple structural properties due to the fact that mechanistic paths need to be causally consistent (i.e., each step of the path needs to be causally linked to the next step).

#### Specifying the overall path

The overall path specification can be done using the JSON Schema developed for INDRA Statements (see https://github.com/sorgerlab/indra/blob/master/indra/resources/statements_schema.json). The *path* is specified via an overall *type*, and, depending on the type, the appropriate Agent arguments.

#### Entity constraints

It is possible to specify constraints on the entities (*entity_constraints*) appearing along the path, for instance, whether to include or exclude certain Agents. The keys for these specifications are *include* and *exclude* respectively.

#### Relationship constraints

It is also possible to specify constraints on relationships along the path (*relationship_constraints*) with the *include* and *exclude* keys.

#### Examples

Example: "How does EGFR lead to ERK phosphorylation without including PI3K or any transcriptional regulation?"

```
{"type": "path_property",
 "path": {
    "type": "Phosphorylation",
    "enz": {
        "type": "Agent",
        "name": "EGFR"
        },
    "sub": {
        "type": "Agent",
        "name": "ERK"
        }
    },
  "entity_constraints": {
    "exclude": [
        {"type": "Agent",
         "name": "PI3K"}
        ]
    },
  "relationship_constratints": {
    "exclude": [
        {"type": "IncreaseAmount"},
        {"type": "DecreaseAmount"}
        ]
    }
 }
```

### 1.4.3 Simple intervention properties

Simple intervention properties focus on the effects of targeted interventions on one or more entities in the model without considering comparisons or optimization across multiple interventions.

#### Specifying an intervention

An intervention can be specified either on a single entity readout or on a path-level effect (we call this a *reference*, i.e., something that the intervention is meant to change). In the first case, the readout is represented, again, as an INDRA Agent, with name, grounding and state. In the second case, a path is represented as and INDRA Statement with type and Agent arguments. The intervention itself is represented as a list of Agents with additional parameters to specify the type of intervetion.

#### Specifying the reference

The *reference* can either have *type* of *relationship* or *entity*. In case of a *relationship*, the specification is an INDRA Statement JSON. In case of an *entity*, the specificaton is an INDRA Agent JSON (see references above).

#### Specifying the intervetion

The *intervention* consists of a list of intervening entities (specified as INDRA Agent JSONs) and the perturbation by which the intervention applies to these entities (i.e., *increase*, *decrease*).

#### Examples

Example: "How does Selumetinib affect phosphorylated MAPK1?"

```
{"type": "simple_intervention_property",
 "reference": {
    "type": "Agent",
    "name": "MAPK1",
    "mods": [
        {"mod_type": "phosphorylation"}
        ]
    },
"intervention": [
    {"entity": {
        "type": "Agent",
        "name": "Selumetinib"
        },
     "perturbation": "increase"
     }
    ]
 }
```

### 1.4.4 Comparative intervention properties

Comparative intervention properties are similar to simple intervention properties but are more general in that they can be used to express comparisons or optimality among a set of possible intervetions. The specification consists, again, of a *reference*, but this time, a list of *interventions* rather than a single *intervention*. The comparison also needs to be specified, i.e., whether the intervetion is meant to *increase* or *decrease* the *reference*.

For comparative intervention properties, the *reference* and each possible *intervention* is specified as above.

### Examples

Example: "Is Selumetinib or Vemurafenib optimal in decreasing ERK activation by EGF?"

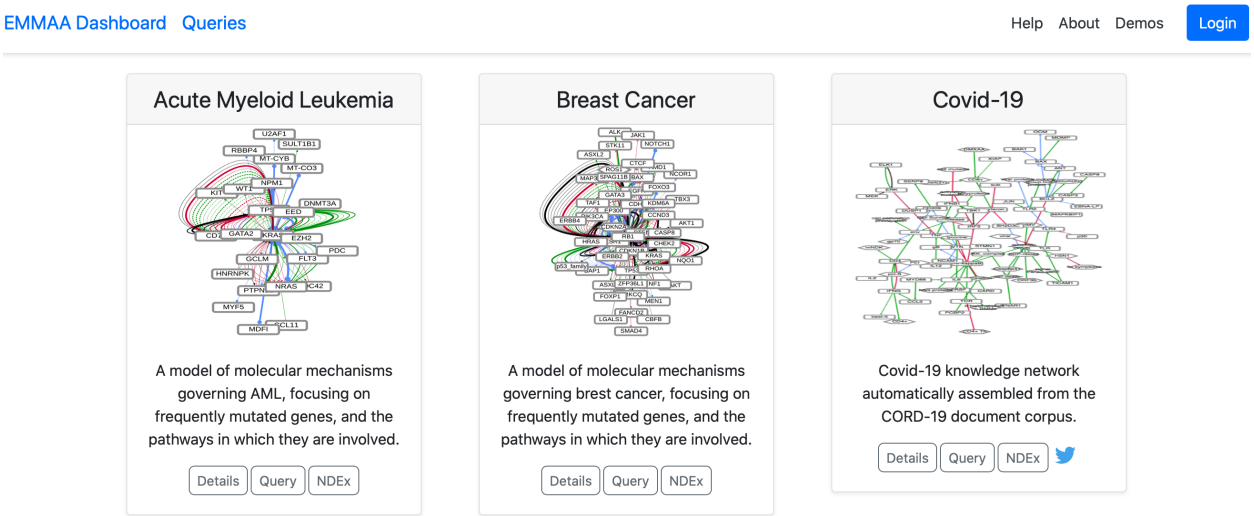
```
{"type": "comparative_intervention_property",
 "reference": {
    "type": "Activation",
    "subj": {
        "name": "EGF",
        },
    "obj": {
        "name": "ERK",
        }
    },
"interventions": [
    [{"entity": {
        "type": "Agent",
        "name": "Selumetinib"
        },
     "perturbation": "increase"
     }],
    [{"entity": {
        "type": "Agent",
        "name": "Vemurafenib"
        },
     "perturbation": "increase"
     }]
    ],
  "comparison": "increase"
 }
```

CHAPTER 2

# EMMAA Dashboard

The EMMAA Dashboard is accessible at http://emmaa.indra.bio.

The EMMAA Dashboard is the main entrypoint for users to interact with models. Each card on the dashboard represents a model. Currently, users can browse and link out to interactive, searchable network views of multiple disease and pathways models, as well as details of the latest tests applied to the models. The user can also navigate to a queries page where queries about the models can be answered. Users are able to sign up for specific notifications about one or more automatically built, tested and analyzed models. Some models also have a Twitter account and a link to it is provided on the dashboard if available




Please read the sections below to learn how different EMMAA pages work.

## 2.1 EMMAA Models Page

The models page contains detailed information about the selected model in four tabs: *Model*, *Tests*, *Papers*, and *Curation*. At the top of the page the selected model is shown in a drop-down menu. Another model can also be selected and loaded from the menu.

### 2.1.1 Link to statement details

To see further details regarding a mechanism, links to a separate page are generated for all statements where possible. To read more about that page, see: *EMMAA Statement Evidence Page*.

Fig. 1: *Link to statement evidence page*

### 2.1.2 Model Tab

The model tab contains model info with the model description, the date the model was last updated and the date when the displayed state of the model was generated. By default the latest available state of the model is displayed but the user has an option to explore earlier states by clicking on an earlier time point on any of the time plots across the tabs (for more details see: *Load Previous State of Model*). Links to the NDEx website where a network view of the model can be examined and to the Twitter account if available are provided. It is possible to download the models in various formats and the corresponding buttons are placed next.

The page also displays properties of the current state of the model, namely, the distribution of statement types, the top 10 agents in the model, the distribution of knowledge sources (reading systems and databases) of model statements and the statements with the most support from various knowledge bases. The table with most supported statements also has a button "All statements" clicking on which a user can be redirected to a page showing all statements in the model: *EMMAA All Statements Page*. Further, the page shows how the number of statements in the model has evolved over time, and which statements were added to the model during the most recent update.

### 2.1.3 Tests Tab

At the top of the tests tab, a drop down menu displays which test corpus was used for the currently displayed test results. Clicking on the drop down menu will display all available test corpora for the current model. Clicking "Load Test Results" will load the test results for the selected test corpus.

The tests tab contains two related plots: one showing the evolution over time of the percentage of applicable tests that passed, and another showing the absolute number of tests that were applied to the model and the number of tests that passed in each of supported model types. For the first few months of the project, the tests were only run on a PySB model assembled from EMMAA model statements. Later three additional model types were added, namely, PyBEL graph, signed directed graph and unsigned directed graph.

Model Tests Papers Curation

## Model Info

| Item | Details |
|---|---|
| Model Description | A model of molecular mechanisms governing AML, focusing on frequently mutated genes, and the pathways in which they are involved. |
| Latest Data Available | 2021-02-03 |
| Data Displayed | 2021-02-03 |
| Network on Ndex | ef58f76d-f6a2-11e8-aaa6-0ac135e8bacf |

## Download as

json jsonl bngl indranet kappa

## Statement Types Distribution

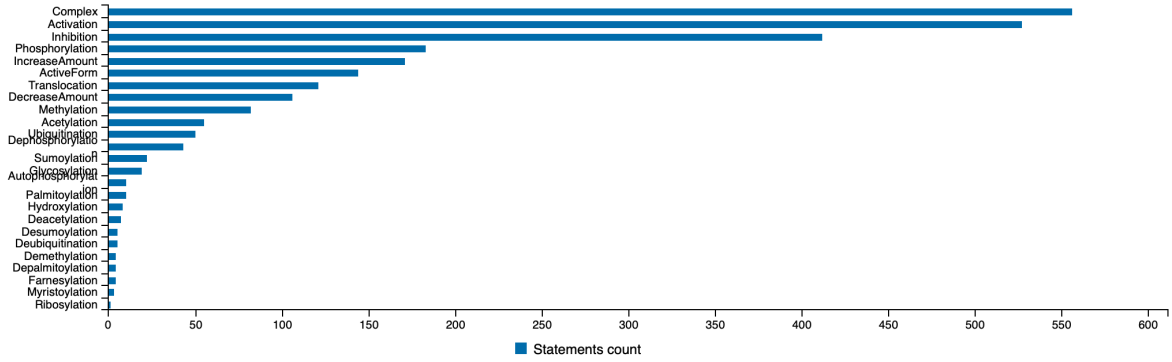



Fig. 2: *The top of the model tab*

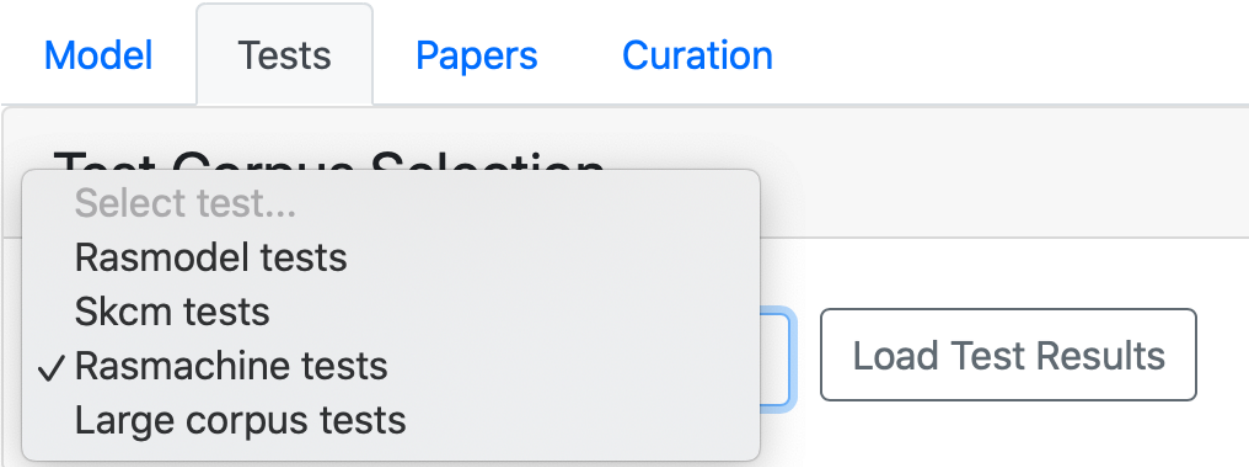



Fig. 3: *The results from different test corpora can be loaded. Here "Rasmodel Tests", "Skcm tests", "Rasmachine Tests", and "Large Corpus Tests" are available.*

If any new tests were applied in the latest test run of the model they are shown under *New Applied Tests*. A green check mark is shown for tests that passed and a red cross is shown for the tests that did not. The marks can be clicked on and link to a detailed test results page where the detailed path(s) or a reason for the model not having passed the test will be shown. To read more about the detailed test results page, see: *EMMAA Detailed Test Results*.

New tests that passed for any of the model types are shown under *New Passed Tests* along with the top path found. The statements supporting the path are can be seen by clicking on a path which links out to the detailed test results page for the test.

Further down, all tests applied to the model are shown. Similarly to new applied tests, this table also contains green and red marks indicating the test status, linking to detailed test results page.

### 2.1.4 Papers Tab

The Papers tab shows statistics for both processed papers and papers that support assembled model statements. At the top of the Papers tab the time series plot shows the changes in the counts of both paper groups over time.

Further down, papers with the largest number of assembled statements are shown. The statements extracted for each paper can be viewed by clicking on a paper title (see: *EMMAA Individual Paper Page*).

Finally, a list of papers processed after the previos update is displayed. The table is sorted first by the number of assembled statements and then by the number of raw statements extracted from the paper. One or both of these numbers can be zero. Zero assembled statements with a positive number of raw statements means that the raw statements were filtered from the model during the assembly process. Two zeros mean that the paper was processed but no statements were extracted from it. The second column in this table provides a link to the original publication as an external resource.

### 2.1.5 Curation Tab

The Curation tab summarizes statistics related to curations for statements that are part of the model. At the top of the tab two barplots show the counts of evidences and assembled statements curated by individual curators.

The next plot shows the number of curations grouped by type.

Finally, the number of curated statements and evidences over time is shown.

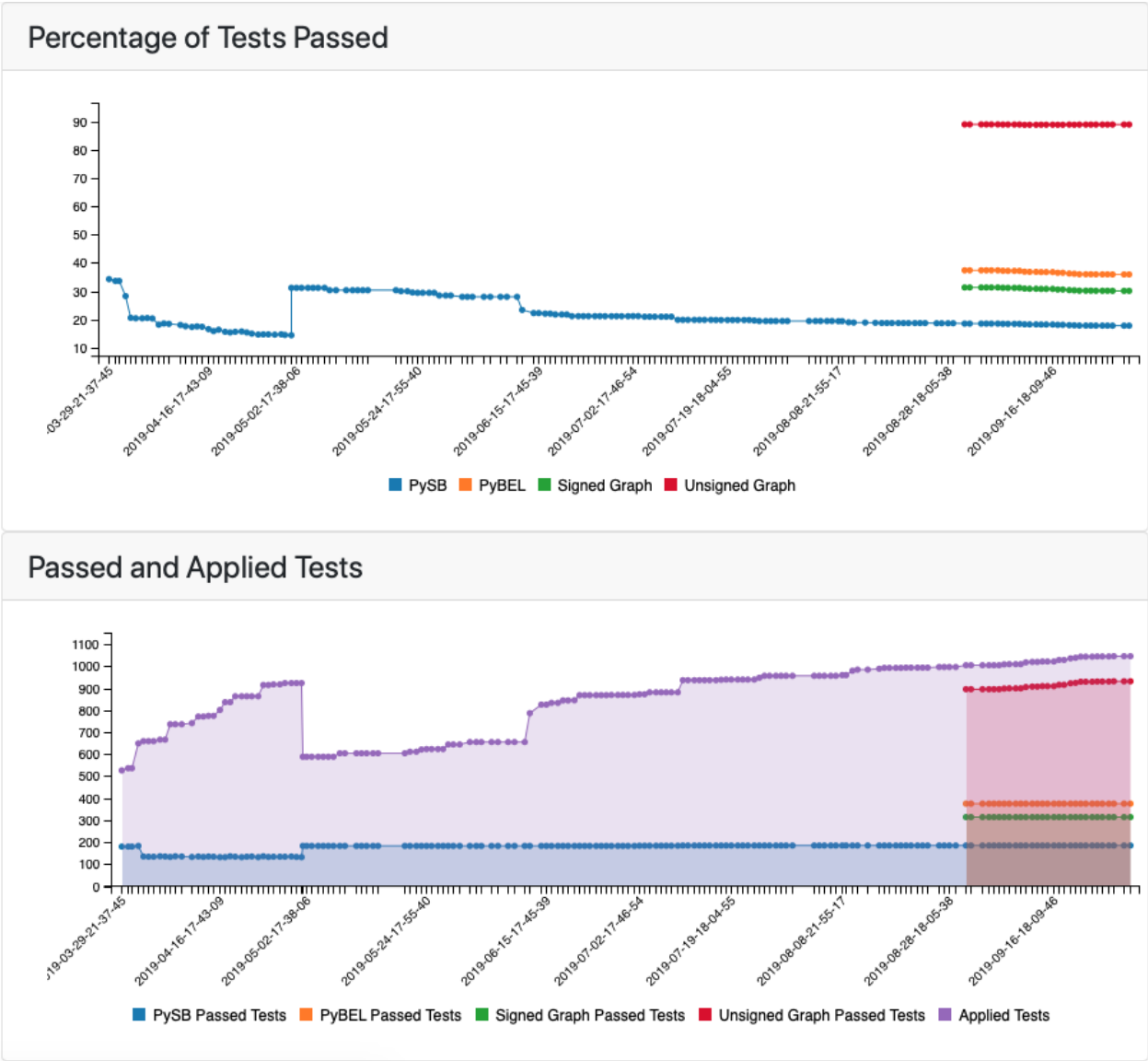



Fig. 4: *The top of the tests tab showing the percentage of tests passed together with applied and passed tests in different model types*

New Applied Tests

| Test | PySB | PyBEL | Signed Graph | Unsigned Graph |
|---|---|---|---|---|
| BMP2 activates CASP9. | ✗ | ✓ | ✓ | ✓ |
| BMP2 increases the amount of CCND3. | ✗ | ✓ | ✓ | ✓ |
| Inflammatory response increases the amount of BMP2. | ✗ | ✗ | ✗ | ✗ |
| BMP2 activates MEK. | ✗ | ✓ | ✓ | ✓ |
| BMP2 activates p38. | ✗ | ✗ | ✗ | ✓ |
| BMP2 inhibits cell cycle. | ✗ | ✗ | ✓ | ✓ |
| BMP2 activates cell differentiation. | ✗ | ✗ | ✓ | ✓ |
| BMP2 activates PKC. | ✗ | ✗ | ✗ | ✗ |
| BMP2 inhibits STAT3. | ✗ | ✓ | ✓ | ✓ |

Fig. 5: *If new tests were applied, they will be shown together with a breakdown of a test status per each model type*

## New Passed Tests

| Test | Top Path |
|---|---|
| **New passed tests for PyBEL model.** | |
| BMP2 activates CASP9. | BMP2 → PTEN → BAX → CASP9 |
| BMP2 increases the amount of CCND3. | BMP2 → PTEN → PIK3CA → CCND3 |
| BMP2 activates MEK. | BMP2 → PTEN → KRAS → MEK |
| BMP2 inhibits STAT3. | BMP2 → PTEN → EGFR → STAT3 |
| **New passed tests for Signed Graph model.** | |
| BMP2 activates CASP9. | BMP2 → PTEN → BAX → CASP9 |
| BMP2 increases the amount of CCND3. | BMP2 → PTEN → PIK3CA → CCND3 |
| BMP2 activates MEK. | BMP2 → PTEN ⊣ EGFR ⊣ MEK |
| BMP2 inhibits cell cycle. | BMP2 → PTEN → RB1 ⊣ cell cycle |
| BMP2 activates cell differentiation. | BMP2 → PTEN → cell differentiation |
| BMP2 inhibits STAT3. | BMP2 → PTEN ⊣ EGFR → STAT3 |
| **New passed tests for Unsigned Graph model.** | |
| BMP2 activates CASP9. | BMP2 → PTEN → BAX → CASP9 |

Fig. 6: *If new tests were passed, they will be shown together with a top path*

## All Test Results

| Test | PySB | PyBEL | Signed Graph | Unsigned Graph |
|---|---|---|---|---|
| AGT activates EGFR. | ✓ | ✓ | ✓ | ✓ |
| AGT activates IGF1R. | ✗ | ✓ | ✓ | ✓ |
| AGT bound to AGTR1 activates ERK. | ✗ | ✗ | ✓ | ✓ |
| Kinase-active AKT activates ESR2. | ✗ | ✗ | ✗ | ✗ |
| Kinase-active AKT activates SP1. | ✗ | ✗ | ✗ | ✗ |

Fig. 7: *Part of the list showing all applied tests with a status indicator for passed/failed*

## Number of Papers over Time

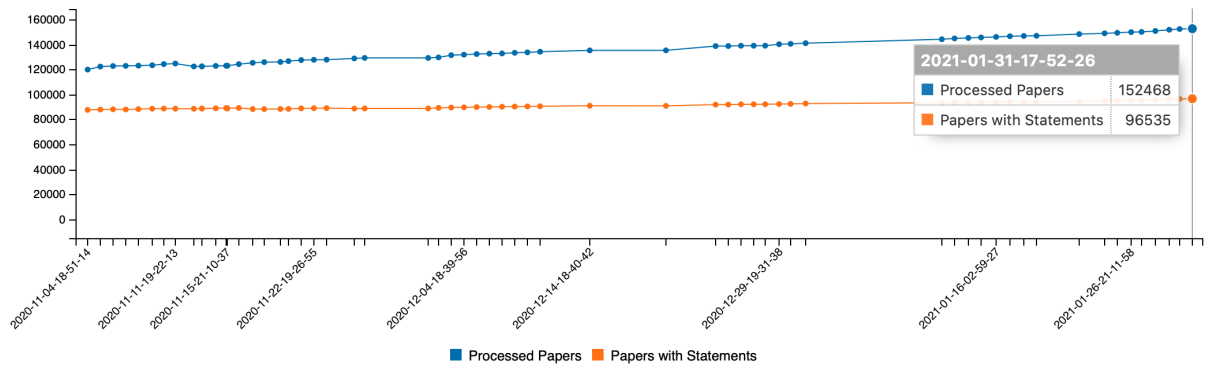



Fig. 8: *Number of processed papers and papers with assembled model statements over time*

## New Papers

| Paper Title | Link | Assembled Statements | Raw Statements |
|---|---|---|---|
| Preclinical evaluation of gilteritinib on NPM1-ALK driven Anaplastic Large Cell Lymphoma Cells. | PubMed | 1 | 9 |
| Incidence of Adverse Cutaneous Reactions to Epidermal Growth Factor Receptor Inhibitors in Patients with Non-Small-Cell Lung Cancer. | PubMed | 1 | 2 |
| Epidermal growth factor receptor tyrosine kinase inhibitor remodels tumor microenvironment by upregulating LAG-3 in advanced non-small-cell lung cancer. | PubMed | 1 | 1 |
| Shikonin Inhibits Cholangiocarcinoma Cell Line QBC939 by Regulating Apoptosis, | PubMed | 0 | 16 |

Fig. 9: *Example of new processed papers table*

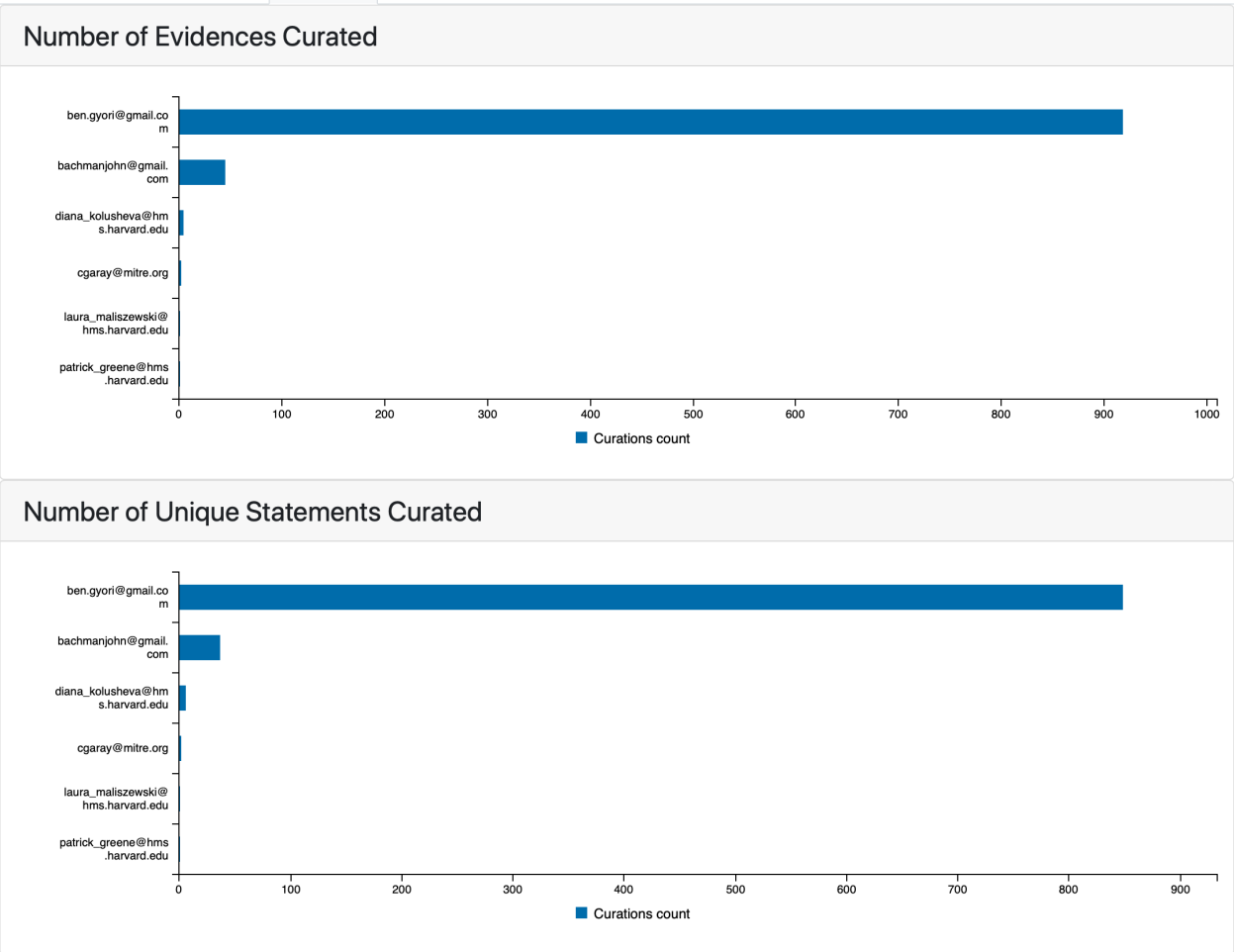



Fig. 10: *Counts of evidences and statements curated by individual curators*

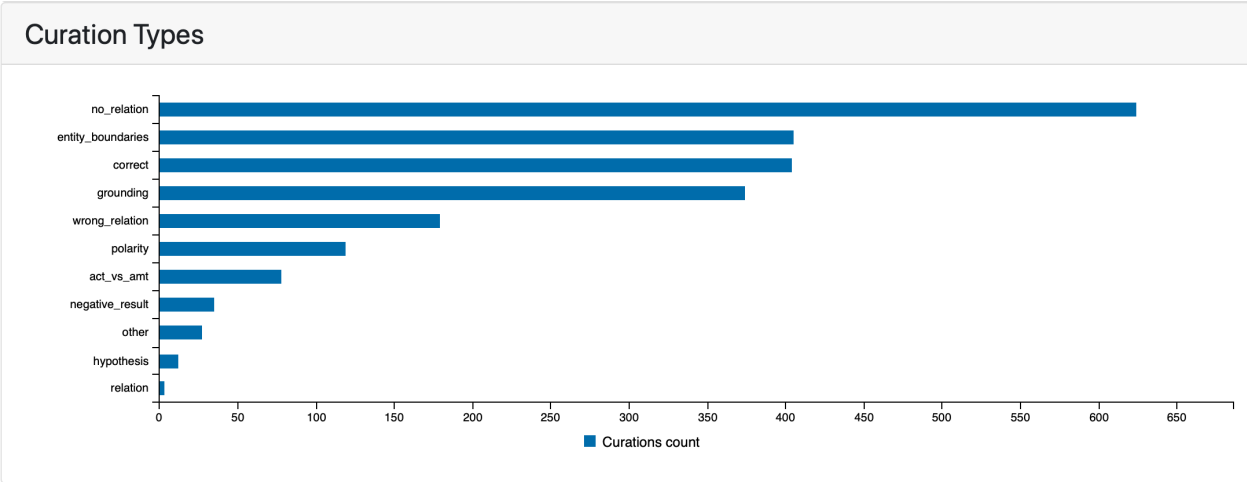



Fig. 11: *Curations grouped by type*

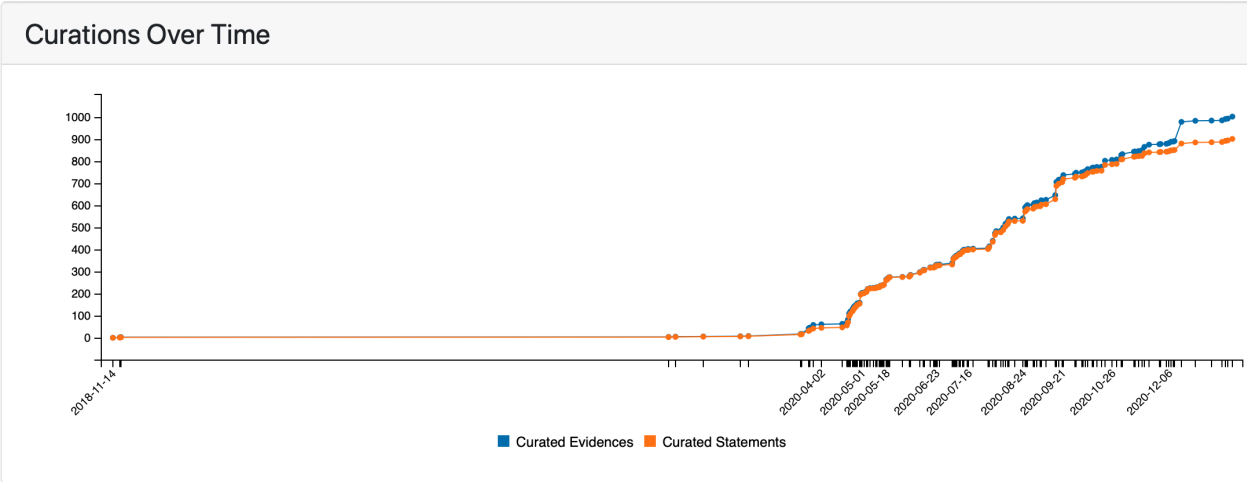



Fig. 12: *Curations over time*

### 2.1.6 Load Previous State of Model

To view the state of the selected model together with the test results for a particualar date, click on any data point for the desired date in any of the time series shown on either the Model tab, the Tests tab or the Papers tab.

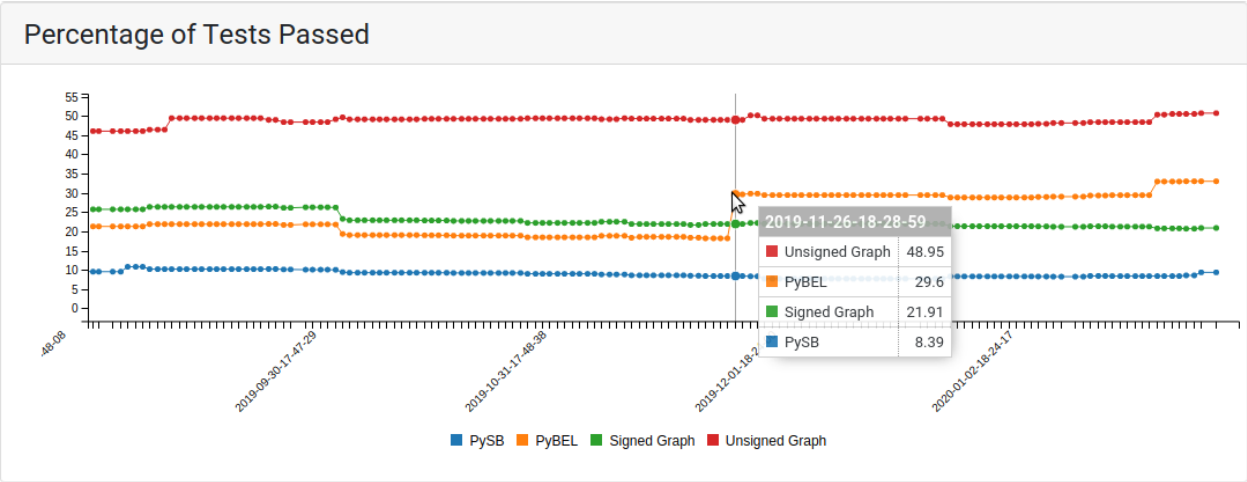



Fig. 13: *Clicking on a data point in any time series will link to the state of the model and the test results for the associated date.*

Clicking the data point will link back to the same models page with data loaded for the selected date. The model info section displays the selected date as well as the date for the most recent data. Any time series show data up to the selected date. Any section showing new updates, such as "New Passed Tests", shows what was new on the selected date while "All Test Results" shows the state of the results were in. Clicking on "Go To Latest" on the top panel will link back to the most recent state of the model.

## 2.2 EMMAA Statement Evidence Page

Any statement displayed on any of the other pages (model page, detailed test or query results) is linked to a statement evidence page where evidences from different sources can be browsed and curated.

Go to Latest | Acute Myeloid Leukemia (AML) | Load Model

Model | Tests

Model Info

| Item | Details |
|---|---|
| Latest Data Available | 2020-01-30 |
| Data Displayed | 2019-11-10 |
| Network on Ndex | ef58f76d-f6a2-11e8-aaa6-0ac135e8bacf |

Fig. 14: *When the state of the model for a previous date is shown, the date is diplayed in "Data Displayed". Clicking on "Go To Latest" on the top panel will link back to the most recent state of the model*

Statement Evidence and Curation

EGFR binds ERBB2. 39 1 — 10/3382 JSON

| | | |
|---|---|---|
| reach | For example, HER-2 binds EGFR, and the kinase activity of HER-2 subsequently phosphorylates the heterodimer, which leads to phosphatidylinositol 3-kinase (PI3K)/Akt and Ras and MEK signaling pathway activation [XREF_BIBR]. | 26728266 |
| reach | HER2 and EGFR binding and displacement of binding by competitors were found for (111) In-bsICs. | 23525982 |
| reach | Pertuzumab, another anti-HER2 humanized monoclonal antibody, binds HER2 at a different epitope of the HER2 extracellular domain from that at which trastuzumab binds, inhibiting not only homodimerization of HER2 but also heterodimerization of HER1 and HER2 and HER2 and HER3 [XREF_BIBR]. | 23346316 |

Fig. 15: *Statement evidence view*

At the top of the table, the statement itself is presented followed by a list of sentences supporting this statement. There are several badges that represent additional information about the statement. The blue badge with a flag in the example above shows have many paths this statement is a part of. A green or a red badge with a pencil shows how many times this statement was curated as correct or incorrect respectively. The grey badge shows the number of loaded evidence and the total number of evidences supporting this statement. Clicking on the JSON badge opens a new page containing the JSON representation of the statement. For each evidence the knowledge source and external link to the publication is given. Clicking on the pencil badge to the left of the evidence, a user can curate this evidence.

## 2.3 EMMAA All Statements Page

The All statements page allows to browse and curate statement evidences similar to the statement evidence page but in this case all statements in the model are listed. By clicking on any statement, a user can open its evidences. By default the statements are sorted by the number of supporting evidence they have, but it is possible to sort them by the number of paths they contribute to. The "Previous" and "Next" buttons allow to page through the full list of statements (only 1000 statements per page are loaded). The "Filter Curated" button allows to filter out the statements that have been already curated. It's also possible to download all statements in JSON format by clicking on "Download Statements." Each statement can have multiple badges that have the same meaning as in the statement evidence page. A blue badge with a flag shows how many paths this statement is a part of. A green or red badge with a pencil shows how many times this statement was curated as correct or incorrect respectively. A grey badge shows the number of loaded evidences and the total number of evidences supporting this statement. Clicking on the JSON badge opens a new page containing the JSON representation of the statement.

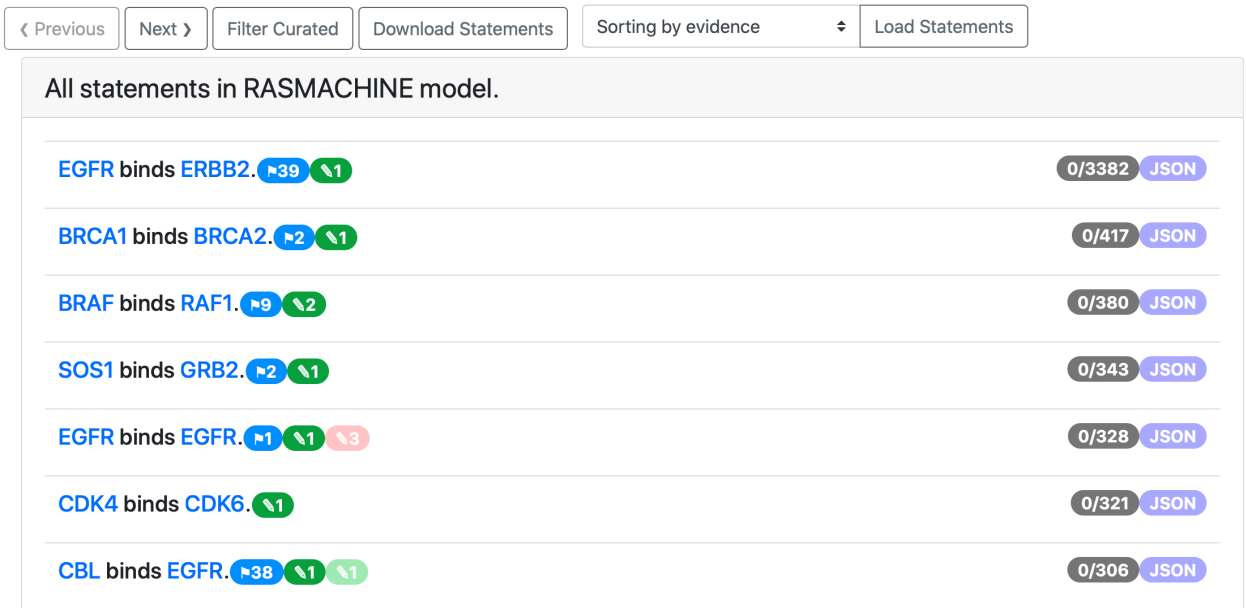



Fig. 16: *All statements page view*

## 2.4 EMMAA Individual Paper Page

By clicking on a paper title on the Papers tab on model page, a user is redirected to an individual paper page that contains model statements from this paper. The view here is similar to the statement evidence or the all statements page with the exception that the statements and evidences are filtered to only those that are extracted from a given

paper. To browse and curate the evidences, a user needs to click on a statement. Each statement can have multiple badges that have the same meaning as in the statement evidence page. A blue badge with a flag shows have many paths this statement is a part of. A green or a red badge with a pencil shows how many times this statement was curated as correct or incorrect respectively. The grey badge shows the number of loaded evidences and the total number of evidences supporting this statement. Clicking on the JSON badge opens a new page containing the JSON representation of the statement.

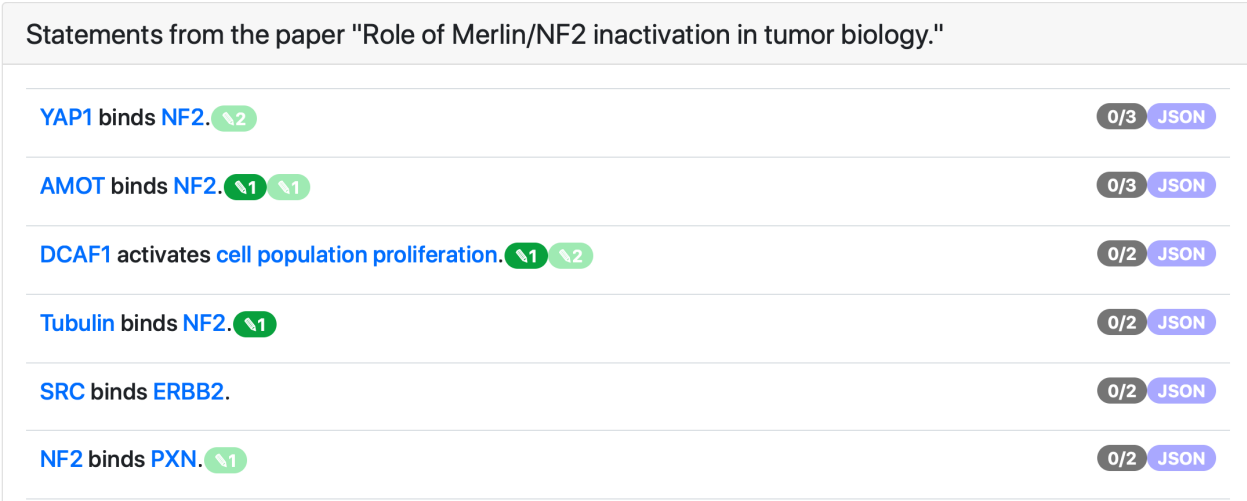



Fig. 17: *Individual paper page view*

## 2.5 EMMAA Model Queries

The Queries page can be accessed by clicking the "Queries" link at the top of the Dashboard website. The page contains the forms to submit queries and results of queries in three tabs *Static*, *Dynamic*, and *Open Search* corresponding to three currently supported query types.

### 2.5.1 Static Queries

This tab allows to submit queries and view the results for static queries that involve finding mechanistic paths to explain causal relation. Queries are run against four model types: PySB, PyBEL, signed graph, and unsigned graph.

#### Submitting a Query

The model queries page can answer direct queries about one or more models. A static query consists of a statement type, a subject, and an object. Together with the query, at least one model needs to be selected for the query submission to be valid.

If the query is badly formatted or missing information, an error will be shown stating the type of error.

#### Viewing the results

The query will be received by the query service and return a response which is displayed in the Results table below. Similarly to the results of tests on the model page, query results are presented as a grid of green, red and grey marks. A green check mark is shown for queries that passed and a red cross is shown for the queries that did not. Grey cirlce

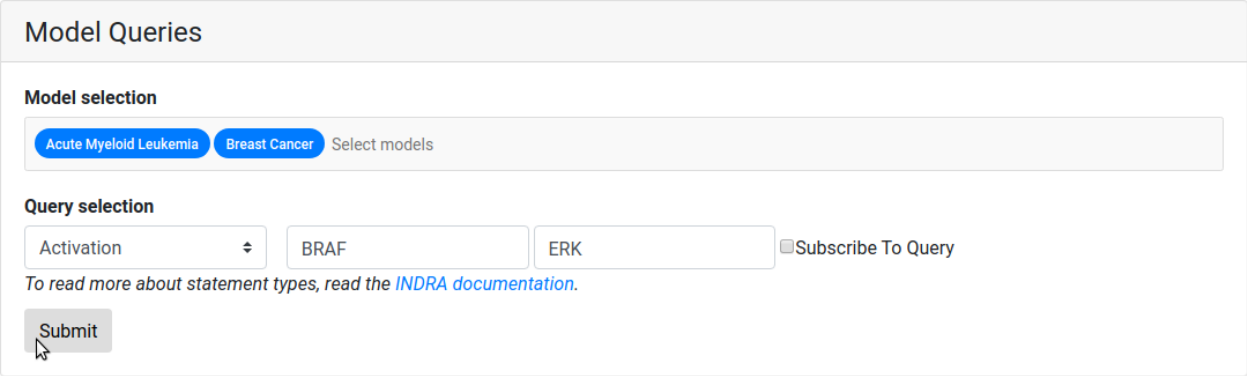



Fig. 18: *The query ready to be submitted that asks if BRAF activates ERK in the AML and BRCA cancer models.*

will be shown for queries not applicable for selected model. The marks can be clicked on and link to a detailed query results page where the detailed path(s) or a reason for the model not having passed the test will be shown.

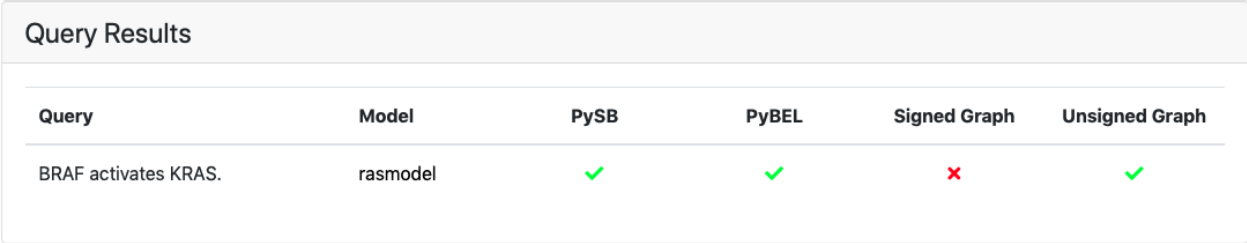



Fig. 19: *The above query resolved, showing the result per model and model type. Detailed results can be viewed by clicking on a green/red mark.*

### 2.5.2 Dynamical Queries

This tab allows to submit and view the results for queries about dynamical model properties. To answer these queries simulations are run on a PySB-assembled EMMAA model.

#### Submitting a Query

Dynamical query requires the user to specify the model, the entity to run simulations for, a temporal pattern and, for some patterns, whether the entity amount should be high or low. An observable entity can be described using natural language, e.g. "phosphorylated MAP2K1".

#### Viewing the results

Results of the dynamical queries include a green/red mark showing whether the required condition was satisfied in more than a half of simulations and a plot of the observable's time cours during the simulation.

### 2.5.3 Open Search Queries

This tab allows submitting and viewing the results of open search queries that involve finding mechanistic paths upstream or downstream of an entity of interest. Similar to static queries, open search queries are run against four model types: PySB, PyBEL, signed graph, and unsigned graph.

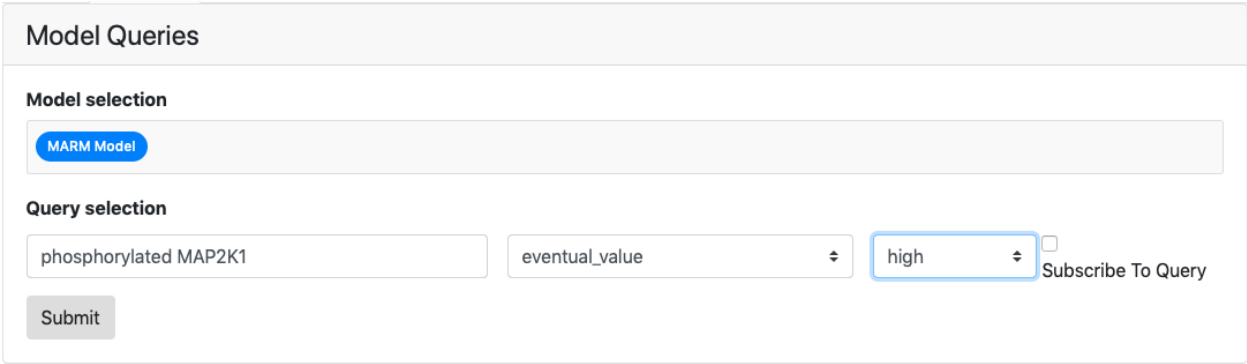



Fig. 20: *The query ready to be submitted that asks whether phosphorylated MAP2K1 is eventually high in the MARM model.*

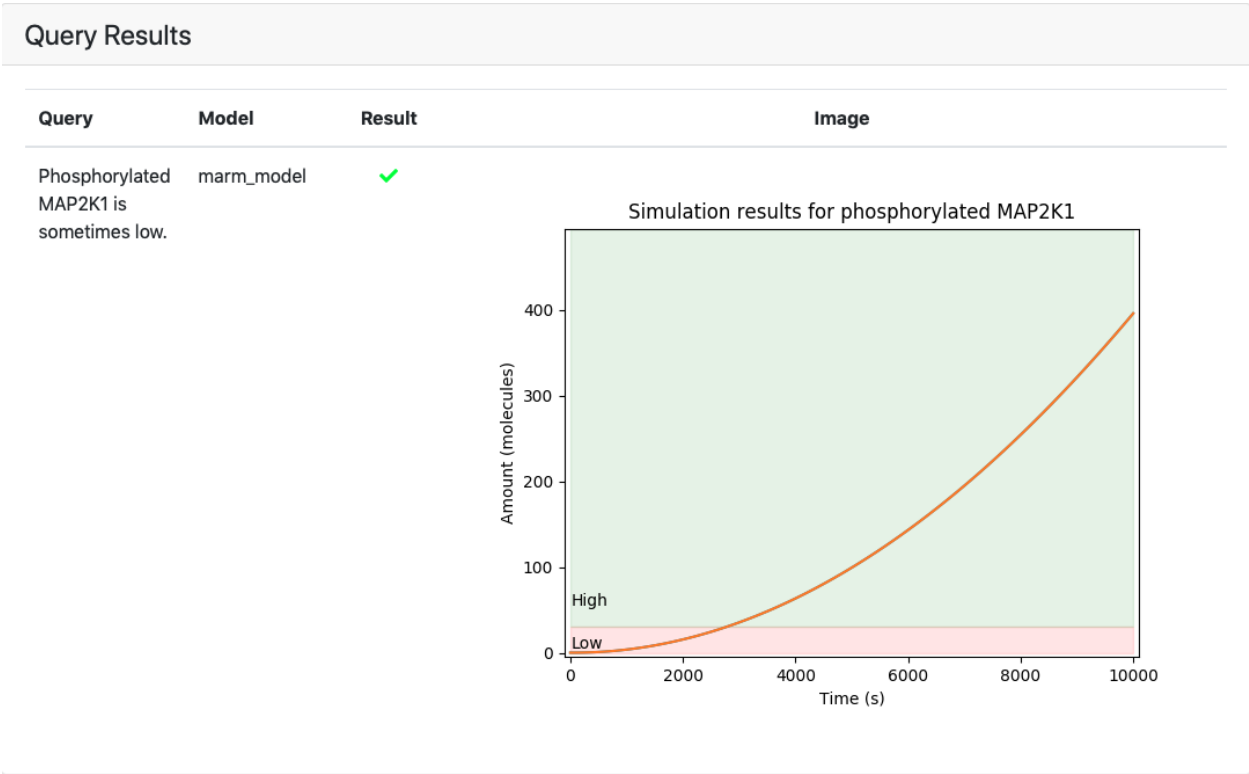



Fig. 21: *The above query resolved, showing the result per model.*

### Submitting a Query

The model queries page can answer direct queries about one or more models. An open search query consists of a statement type, an agent (entity), and an agent's role (subject for downstream search and object for upstream search). Optionally, a user can also limit the search results to only include paths to or from genes/proteins, small molecules, or biological processes. Together with the query, at least one model needs to be selected for the query submission to be valid.

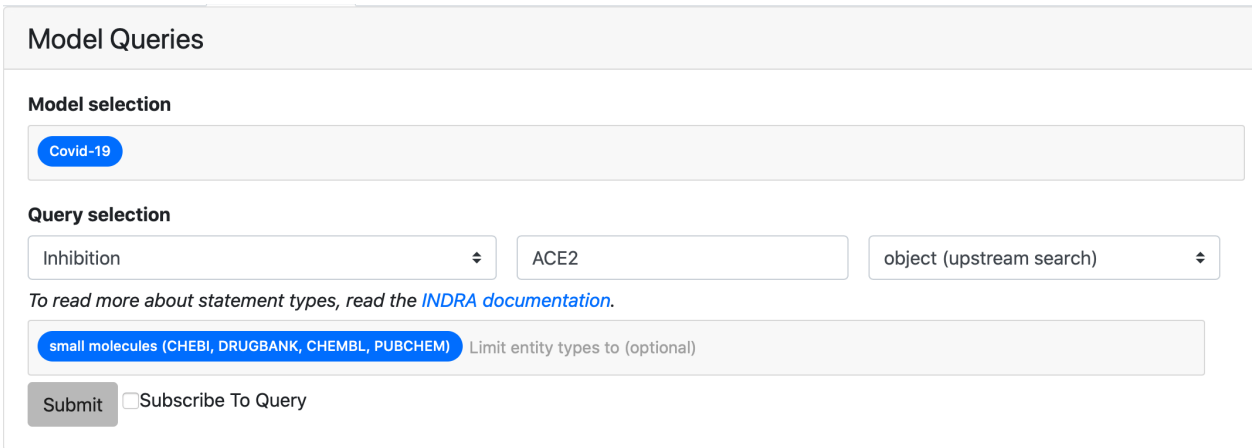
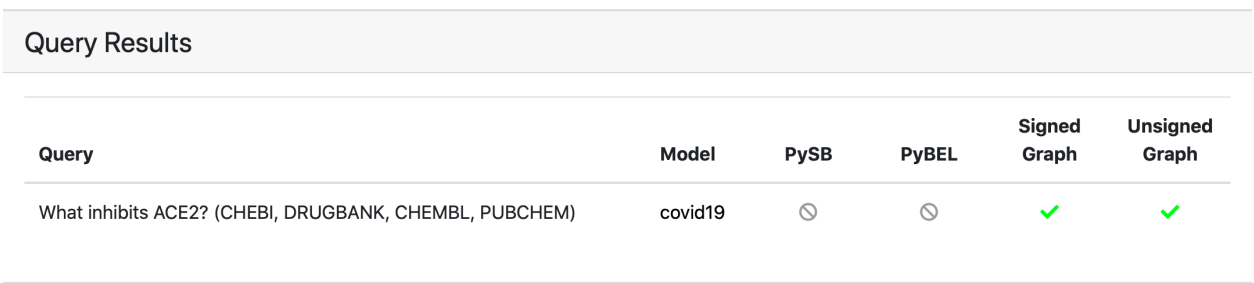



Fig. 22: *The query ready to be submitted that asks what small molecules inhibit ACE2 in Covid-19 model*

If the query is badly formatted or is missing information, an error will be shown stating the type of error.

### Viewing the results

The query is received by the query service which returns a response in a format similar to the result of static queries.




| Query | Model | PySB | PyBEL | Signed Graph | Unsigned Graph |
|---|---|---|---|---|---|
| What inhibits ACE2? (CHEBI, DRUGBANK, CHEMBL, PUBCHEM) | covid19 | ⊘ | ⊘ | ✓ | ✓ |

Fig. 23: *The above query resolved, showing the result per model and model type. Detailed results can be viewed by clicking on a green/red mark. Grey circles mean that these model types are not available for a selected model.*

## 2.5.4 Waiting for results

For either of the query types the page displays "Waiting for server response" and a loader bar while the query is being executed. The typical response time can be up to a minute so please be patient when posting queries.

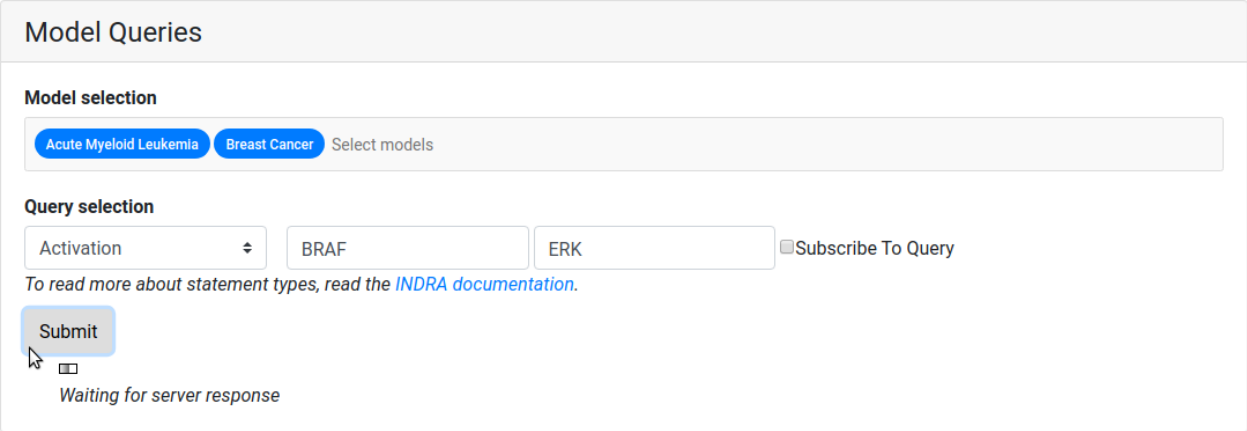
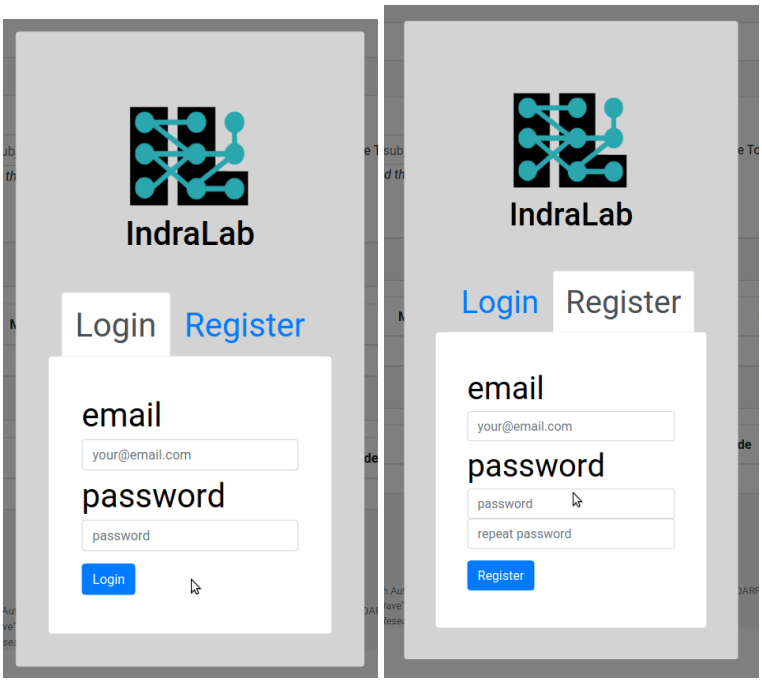



Fig. 24: *While the query resolves, a small animation is shown.*

### 2.5.5 Logging In and Registering a User

A user can log in by clicking the "Login" button to the right on the navigation bar. When clicking the login button, an overlay shows up asking for credentials. A user can also create an account by clicking "Register" if they don't already have an account.




*The login and registration tabs of the login overlay.*

### 2.5.6 Subscribing to a Query

When logged in, a user can register a query for subscription. To register a subscription to a query, the tick box for "Subscribe To Query" has to be ticked when the query is submitted. Both static and dynamic queries can be subscribed to. After submission, the query is associated with the logged in user. When returning to the page, the subscribed queries will be loaded together with their latest results.

Subscribed Queries

| Query | Model | PySB | PyBEL | Signed Graph | Unsigned Graph |
|---|---|---|---|---|---|
| FLT3 activates KRAS. | aml | ✔ | ✔ | ✔ | ✔ |

Fig. 25: *The table for subscribed queries, here for the query Activation(FLT3, KRAS) of the AML cancer model.*

### 2.5.7 Email Notifications of Subscribed Queries

If a user subscribes to a query, they are also signed up for daily email updates that will be sent out if there is an update to any of the subscribed queries. An update to a query is defined as there being a change in the associated model that answers the query. The email lists the updates by query type, query, model and model type. If are no updates for one of the query types, only the query type that has any updates will be shown. For static queries, a direct link to the detailed query results is provided.

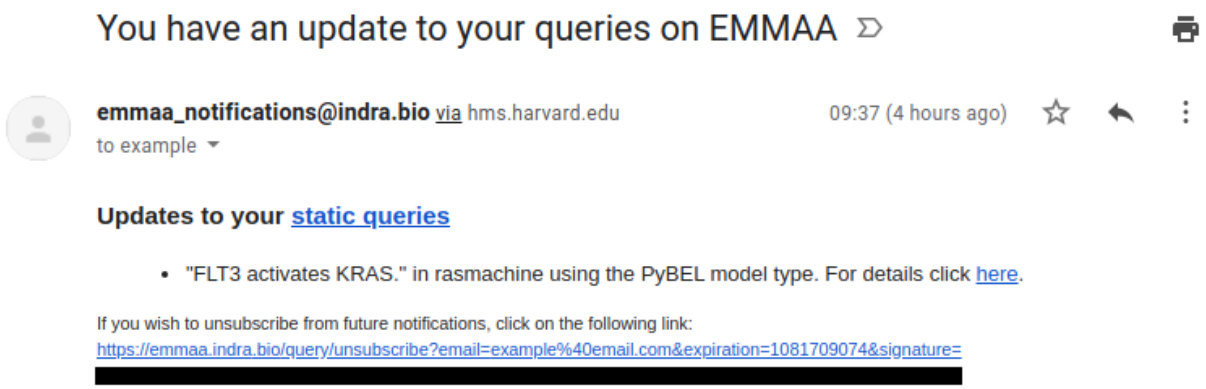



Fig. 26: *An example of an email notification for a query. Here, an update to the query Activation(FLT3, KRAS) of the Ras Machine model of the PyBEL model type is shown under "static queries". The unsubscribe link at the bottom links out to the unsubscribe page (see below).*

#### Unsubscribing From Query Notifications

In every email notification there is an unsubscribe link in the footer of the email. To unsubscribe from queries, follow the link to the unsubscribe page. On the unsubscribe page, all active subscriptions for the associated email are shown with tick boxes for each subscription and one tick box for unsubscribing from all subscribe queries. After ticking the appropriate boxes and submitting the unsubscribe request, a message will be shown describing the status of the request once it resolves.

## 2.6 Test and query result interpretation

Model tests and queries can sometimes fail to produce an explicit result (i.e., a sequence of mechanisms constituting an answer to a query). There are several possible reasons for this. Below, we explain the various result "codes" that can appear on the model tests and query pages.

## Pick queries to unsubscribe from

| query | type | unsubscribe |
|---|---|---|
| FLT3 activates KRAS. for model aml | path property | ☐ |
| FLT3 activates KRAS. for model rasmachine | path property | ☐ |
| SUMO1 activates TP53. for model aml | path property | ☐ |
| BRAF phosphorylates MAP2K1. for model marm_model | path property | ☐ |
| Resveratrol activates inflammatory response. for model painmachine | path property | ☐ |
| Phosphorylated MAP2K1 is sometimes low. for model marm_model | dynamic property | ☐ |
| Phosphorylated MAP2K1 is eventually high. for model marm_model | dynamic property | ☐ |
| *Unsubscribe from all* | | ☐ |

Unsubscribe

Fig. 27: *An example of how the unsubscribe page looks like. All subscribed queries for a given user is shown. Each query can be individually marked for unsubscription. All queries can be unsubscribed simultaneously by ticking the box for "unsubscribe from all"*

- **Path found but exceeds search depth** - Path is found, but the search depth is reached. Search depth is the maximum number of steps taken to reach the object from the subject in the graph representation of the model.
- **Statement subject not in model** - The subject of the query or statement doesn't exist in the model.
- **Statement object state not in model** - The object state of the query or statement does not exist in the model.
- **Query is not applicable for this model** - Only used for queries.
- **No path found that satisfies the test statement** - Only used for tests.
- **Statement type not handled** - The statement type is not valid. Currently supported types:
  - Activation
  - Inhibition
  - IncreaseAmount
  - DecreaseAmount
  - Acetylation
  - Farnesylation
  - Geranylgeranylation
  - Glycosylation
  - Hydroxylation
  - Methylation
  - Myristoylation
  - Palmitoylation
  - Phosphorylation
  - Ribosylation
  - Sumoylation
  - Ubiquitination
  - Deacetylation
  - Defarnesylation
  - Degeranylgeranylation
  - Deglycosylation
  - Dehydroxylation
  - Demethylation
  - Demyristoylation
  - Depalmitoylation
  - Dephosphorylation
  - Deribosylation
  - Desumoylation
  - Deubiquitination

## 2.7 EMMAA Detailed Test Results

The detailed test results page shows a test result at in high detail for a specific model and model type. The left column describes the paths found that satisfies the test. Note that the same test/query can be explained with multiple different paths. The right column contains a detailed description of each edge in the path with a list of english representation of the statements supporting the edge. If a test did not pass, a message explaining why it did not pass is shown.

Query: "FLT3 activates KRAS." for AML (PyBEL) on 2020-01-31-20-18-53

| Path | Support |
|---|---|
| FLT3 → NPM1 → CDC42 → KRAS | **FLT3 → NPM1**<br>FLT3 bound to NPM1 has a component NPM1.<br>**NPM1 → CDC42**<br>NPM1 activates CDC42.<br>**CDC42 → KRAS**<br>CDC42 activates KRAS. |
| FLT3 → NPM1 → KRAS → KRAS | **FLT3 → NPM1**<br>FLT3 bound to NPM1 has a component NPM1.<br>**NPM1 → KRAS**<br>NPM1 is a part of KRAS bound to NPM1.<br>**KRAS → KRAS**<br>KRAS bound to NPM1 has a component KRAS. |

Fig. 28: *The detailed test results for "FLT3 activates KRAS". The left column displays the two paths that satisfy the test for the model and model type. The right column gives detailed information for each of the edges, including its support, for each path.*

### 2.7.1 Results for Different Model Types

The navigation bar contains a drop down menu where another model type can be selected. After selecting the model type to switch to, click on "Load Type" to load the same model test with the selected model type. *Note that only model types available for the specific model are available in the menu.*

PySB Load Type
PySB
PyBEL
Signed Graph
Unsigned Graph

Test: " Phosphatase-active PTEN inhibits MTOR. " for RASMODEL (PySB)

| Path | Support |
|---|---|
| PTEN ⊣ phosphatidylinositol trisphosphate → AKT1 → MTOR | **PTEN ⊣ phosphatidylinositol trisphosphate**<br>PTEN decreases the amount of phosphatidylinositol trisphosphate.<br>**PDPK1 → AKT1**<br>Active PDPK1 phosphorylates AKT1 bound to phosphatidylinositol trisphosphate on T308.<br>**AKT1 → MTOR**<br>Active AKT1 phosphorylates MTOR on S2448. |

Fig. 29: *The drop down menu shows the other available model types for the test on the model.*

### 2.7.2 Non-passing Tests

When a test fails, the detailed test page show a message that describes why the test failed instead of results.

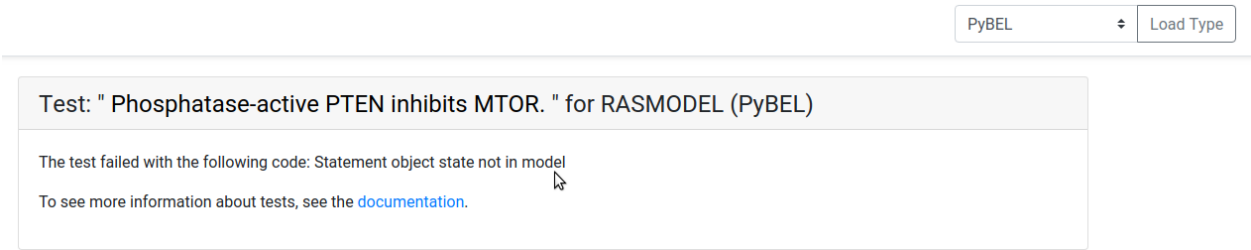



Fig. 30: *The test did not pass and a message is shown describing why.*

CHAPTER 3

# EMMAA modules reference

## 3.1 EMMAA Statement (emmaa.statements)

**class** emmaa.statements.**EmmaaStatement**(*stmt*, *date*, *search_terms*, *metadata=None*)

Bases: object

Represents an EMMAA Statement.

**Parameters**

- **stmt** (*indra.statements.Statement*) – An INDRA Statement
- **date** (*datetime*) – A datetime object that is attached to the Statement. Typically represents the time at which the Statement was created.
- **search_terms** (*list[emmaa.priors.SearchTerm]*) – The list of search terms that led to the creation of the Statement.
- **metadata** (*dict*) – Additional metadata for the statement.

emmaa.statements.**add_emmaa_annotations**(*indra_stmt*, *annotation*)

Add EMMAA annotations to inner INDRA statement.

emmaa.statements.**check_stmt**(*stmt*, *conditions*, *evid_policy='any'*)

Decide whether a statement meets the conditions.

**Parameters**

- **stmt** (*indra.statements.Statement*) – INDRA Statement that should be checked for conditions.
- **conditions** (*dict*) – Conditions represented as key-value pairs that statements' metadata can be compared to. NOTE if there are multiple conditions provided, the function will require that all conditions are met to return True.
- **evid_policy** (*str*) – Policy for checking statement's evidence objects. If 'all', then the function returns True only if all of statement's evidence objects meet the conditions. If

'any', the function returns True as long as at least one of statement's evidences meets the conditions.

**Returns** **meets_conditions** – Whether the Statement meets the conditions.

**Return type** bool

emmaa.statements.**filter_emmaa_stmts_by_metadata**(*estmts*, *conditions*)

Filter EMMAA statements to those where conditions are met.

**Parameters**

- **estmts** (*list[emmaa.statements.EmmaaStatement]*) – A list of EMMAA Statements to filter.
- **conditions** (*dict*) – Conditions to filter on represented as key-value pairs that statements' metadata can be compared to. NOTE if there are multiple conditions provided, the function will require that all conditions are met to keep a statement.

**Returns** **estmts_out** – A list of EMMAA Statements which meet the conditions.

**Return type** list[*emmaa.statements.EmmaaStatement*]

emmaa.statements.**filter_indra_stmts_by_metadata**(*stmts*, *conditions*, *evid_policy='any'*)

Filter INDRA statements to those where conditions are met.

**Parameters**

- **stmts** (*list[indra.statements.Statement]*) – A list of INDRA Statements to filter.
- **conditions** (*dict*) – Conditions to filter on represented as key-value pairs that statements' metadata can be compared to. NOTE if there are multiple conditions provided, the function will require that all conditions are met to keep a statement.
- **evid_policy** (*str*) – Policy for checking statement's evidence objects. If 'all', then the statement is kept only if all of it's evidence objects meet the conditions. If 'any', the statement is kept as long as at least one of its evidences meets the conditions.

**Returns** **stmts_out** – A list of INDRA Statements which meet the conditions.

**Return type** list[indra.statements.Statement]

emmaa.statements.**is_internal**(*stmt*)

Check if statement has any internal evidence.

emmaa.statements.**to_emmaa_stmts**(*stmt_list*, *date*, *search_terms*, *metadata=None*)

Make EMMAA statements from INDRA Statements with the given metadata.

## 3.2 EMMAA Model (emmaa.model)

**class** emmaa.model.**EmmaaModel**(*name*, *config*, *paper_ids=None*)

Bases: object

Represents an EMMAA model.

**Parameters**

- **name** (*str*) – The name of the model.
- **config** (*dict*) – A configuration dict that is typically loaded from a YAML file.

- **paper_ids** (*list(str) or None*) – A list of paper IDs used to get statements for the current state of the model. With new reading results, new paper IDs will be added. If not provided, initial set will be derived from existing statements.

**stmts**
A list of EmmaaStatement objects representing the model

**Type** list[emmaa.EmmaaStatement]

**assembly_config**
Configurations for assembling the model.

**Type** dict

**test_config**
Configurations for running tests on the model.

**Type** dict

**reading_config**
Configurations for reading the content.

**Type** dict

**query_config**
Configurations for running queries on the model.

**Type** dict

**search_terms**
A list of SearchTerm objects containing the search terms used in the model.

**Type** list[*emmaa.priors.SearchTerm*]

**ndex_network**
The identifier of the NDEx network corresponding to the model.

**Type** str

**assembled_stmts**
A list of assembled INDRA Statements

**Type** list[indra.statements.Statement]

**add_paper_ids**(*initial_ids*, *id_type='pmid'*)
Convert if needed and save paper IDs.

**Parameters**

- **initial_ids** (*set(str)*) – A set of paper IDs.
- **id_type** (*str*) – What type the given IDs are (e.g. pmid, doi, pii). All IDs except for PIIs will be converted into TextRef IDs before saving.

**add_statements**(*stmts*)
Add a set of EMMAA Statements to the model

**Parameters** **stmts** (*list[emmaa.EmmaaStatement]*) – A list of EMMAA Statements to add to the model

**assemble_dynamic_pysb**(***kwargs*)
Assemble a version of a PySB model for dynamic simulation.

**assemble_pybel**(*mode='local'*, *bucket='emmaa'*)
Assemble the model into PyBEL and return the assembled model.

**assemble_pysb**(*mode='local'*, *bucket='emmaa'*)
Assemble the model into PySB and return the assembled model.

**assemble_signed_graph**(*mode='local'*, *bucket='emmaa'*)
Assemble the model into signed graph and return the assembled graph.

**assemble_unsigned_graph**(***kwargs*)
Assemble the model into unsigned graph and return the assembled graph.

**eliminate_copies**()
Filter out exact copies of the same Statement.

**extend_unique**(*estmts*)
Extend model statements only if it is not already there.

**get_assembled_entities**()
Return a list of Agent objects that the assembled model contains.

**get_entities**()
Return a list of Agent objects that the model contains.

**get_indra_stmts**()
Return the INDRA Statements contained in the model.

**Returns** The list of INDRA Statements that are extracted from the EMMAA Statements.

**Return type** list[indra.statements.Statement]

**get_new_readings**(*date_limit=10*)
Search new literature, read, and add to model statements

**get_paper_ids_from_stmts**(*stmts*)
Get initial set of paper IDs from a list of statements.

**Parameters** **stmts** (*list[emmaa.statements.EmmaaStatement]*) – A list of EMMAA statements to create the mappings from.

**classmethod load_from_s3**(*model_name*, *bucket='emmaa'*)
Load the latest model state from S3.

**Parameters** **model_name** (*str*) – Name of model to load. This function expects the latest model to be found on S3 in the emmaa bucket with key 'models/{model_name}/model_{date_string}', and the model config file at 'models/{model_name}/config.json'.

**Returns** Latest instance of EmmaaModel with the given name, loaded from S3.

**Return type** *emmaa.model.EmmaaModel*

**run_assembly**()
Run INDRA's assembly pipeline on the Statements.

**save_to_s3**(*bucket='emmaa'*)
Dump the model state to S3.

**static search_biorxiv**(*collection_id*, *date_limit*)
Search BioRxiv within date_limit.

**Parameters**

- **date_limit** (*int*) – The number of days to search back from today.
- **collection_id** (*str*) – ID of a collection to search BioArxiv for.

**Returns** **terms_to_dois** – A dict representing biorxiv collection ID as key and DOIs returned by search as values.

**Return type** dict

**static search_elsevier**(*search_terms*, *date_limit*)

Search Elsevier for given search terms.

**Parameters**

- **search_terms** (*list[emmaa.priors.SearchTerm]*) – A list of SearchTerm objects to search PubMed for.
- **date_limit** (*int*) – The number of days to search back from today.

**Returns** **terms_to_piis** – A dict representing given search terms as keys and PIIs returned by searches as values.

**Return type** dict

**search_literature**(*lit_source*, *date_limit=None*)

Search for the model's search terms in the literature.

**Parameters** **date_limit** (*Optional[int]*) – The number of days to search back from today.

**Returns** **ids_to_terms** – A dict representing all the literature source IDs (e.g., PMIDs or PIIS) returned by the searches as keys, and the search terms for which the given ID was produced as values.

**Return type** dict

**static search_pubmed**(*search_terms*, *date_limit*)

Search PubMed for given search terms.

**Parameters**

- **search_terms** (*list[emmaa.priors.SearchTerm]*) – A list of SearchTerm objects to search PubMed for.
- **date_limit** (*int*) – The number of days to search back from today.

**Returns** **terms_to_pmids** – A dict representing given search terms as keys and PMIDs returned by searches as values.

**Return type** dict

**to_json**()

Convert the model into a json dumpable dictionary

**update_from_disease_map**(*disease_map_config*)

Update model by processing MINERVA Disease Map.

Relevant part of reading config should look similar to:

**{"disease_map": {** "map_name": "covid19map", "filenames" : "all", # or a list of filenames "metadata": {

"internal": true }

}

}

**update_from_files**(*files_config*)

Add custom statements from files.

Relevant part of reading config should look similar to:

**{"other_files": [**

**{** "bucket": "indra-covid19", "filename": "ctd_stmts.pkl", "metadata": {"internal": true, "curated": true}

}

**update_to_ndex**()

Update assembled model as CX on NDEx, updates existing network.

**update_with_cord19**(*cord19_config*)

Update model with new CORD19 dataset statements.

Relevant part of reading config should look similar to:

**{"cord19_update": {**

**"metadata": {** "internal": true, "curated": false },

"date_limit": 5 }

}

**upload_to_ndex**()

Upload the assembled model as CX to NDEx, creates new network.

emmaa.model.**get_assembled_statements**(*model*, *date=None*, *bucket='emmaa'*)

Load and return a list of assembled statements.

**Parameters**

- **model** (*str*) – A name of a model.
- **date** (*str or None*) – Date in "YYYY-MM-DD" format for which to load the statements. If None, loads the latest available statements.
- **bucket** (*str*) – Name of S3 bucket to look for a file. Defaults to 'emmaa'.

**Returns**

- **stmts** (*list[indra.statements.Statement]*) – A list of assembled statements.
- **latest_file_key** (*str*) – Key of a file with statements on s3.

emmaa.model.**get_model_stats**(*model*, *mode*, *tests=None*, *date=None*, *extension='.json'*, *n=0*, *bucket='emmaa'*)

Gets the latest statistics for the given model

**Parameters**

- **model** (*str*) – Model name to look for
- **mode** (*str*) – Type of stats to generate (model or test)
- **tests** (*str*) – A name of a test corpus. Default is large_corpus_tests.
- **date** (*str or None*) – Date for which the stats will be returned in "YYYY-MM-DD" format.
- **extension** (*str*) – Extension of the file.
- **n** (*int*) – Index of the file in list of S3 files sorted by date (0-indexed).
- **bucket** (*str*) – Name of bucket on S3.

**Returns** **model_data** – The json formatted data containing the statistics for the model

**Return type** json

`emmaa.model.last_updated_date(model, file_type='model', date_format='date', tests='large_corpus_tests', extension='.pkl', n=0, bucket='emmaa')`

Find the most recent or the nth file of given type on S3 and return its creation date.

Example file name: models/aml/model_2018-12-13-18-11-54.pkl

**Parameters**

- **model** (*str*) – Model name to look for
- **file_type** (*str*) – Type of a file to find the latest file for. Accepted values: 'model', 'test_results', 'model_stats', 'test_stats'.
- **date_format** (*str*) – Format of the returned date. Accepted values are 'datetime' (returns a date in the format "YYYY-MM-DD-HH-mm-ss") and 'date' (returns a date in the format "YYYY-MM-DD"). Default is 'date'.
- **extension** (*str*) – The extension the model file needs to have. Default is '.pkl'
- **n** (*int*) – Index of the file in list of S3 files sorted by date (0-indexed).
- **bucket** (*str*) – Name of bucket on S3.

**Returns** **last_updated** – A string of the selected format.

**Return type** str

`emmaa.model.load_config_from_s3(model_name, bucket='emmaa')`

Return a JSON dict of config settings for a model from S3.

**Parameters** **model_name** (*str*) – The name of the model whose config should be loaded.

**Returns** **config** – A JSON dictionary of the model configuration loaded from S3.

**Return type** dict

`emmaa.model.load_stmts_from_s3(model_name, bucket='emmaa')`

Return the list of EMMAA Statements constituting the latest model.

**Parameters** **model_name** (*str*) – The name of the model whose config should be loaded.

**Returns** **stmts** – The list of EMMAA Statements in the latest model version.

**Return type** list of emmaa.statements.EmmaaStatement

`emmaa.model.save_config_to_s3(model_name, config, bucket='emmaa')`

Upload config settings for a model to S3.

**Parameters**

- **model_name** (*str*) – The name of the model whose config should be saved to S3.
- **config** (*dict*) – A JSON dict of configurations for the model.

## 3.3 EMMAA Model Test Framework (`emmaa.model_tests`)

This module implements the object model for EMMAA model testing.

**class** `emmaa.model_tests.EmmaaTest`

Bases: `object`

Represent an EMMAA test condition

**get_entities**()
Return a list of entities that the test checks for.

**class** emmaa.model_tests.**ModelManager**(*model*, *mode='local'*)
Bases: object

Manager to generate and store properties of a model and relevant tests.

**Parameters**

- **model** (emmaa.model.EmmaaModel) – EMMAA model
- **mode** (*str*) – If 'local' (default), does not save any exports/images to S3. It is only set to 's3' mode in update_model_manager.py script.

**mc_mapping**
A dictionary mapping a ModelChecker type to a corresponding method for assembling the model and a ModelChecker class.

**Type** dict

**mc_types**
A dictionary in which each key is a type of a ModelChecker and value is a dictionary containing an instance of a model, an instance of a ModelChecker and a list of test results.

**Type** dict

**entities**
A list of entities of EMMAA model.

**Type** list[indra.statements.agent.Agent]

**applicable_tests**
A list of EMMAA tests applicable for given EMMAA model.

**Type** list[*emmaa.model_tests.EmmaaTest*]

**date_str**
Time when this object was created.

**Type** str

**path_stmt_types**
A dictionary mapping statement hashes to a count of paths they are in.

**Type** dict

**add_result**(*mc_type*, *result*)
Add a result to a list of results.

**add_test**(*test*)
Add a test to a list of applicable tests.

**answer_dynamic_query**(*query*, *bucket='emmaa'*)
Answer user query by simulating a PySB model.

**answer_intervention_query**(*query*, *bucket='emmaa'*)
Answer user intervention query by simulating a PySB model.

**answer_open_query**(*query*)
Answer user open search query with found paths.

**answer_path_query**(*query*)
Answer user query with a path if it is found.

**answer_queries**(*queries*, ***kwargs*)
Answer all queries registered for this model.

**Parameters** **queries** (*list[emmaa.queries.Query]*) – A list of queries to run.

**Returns** **responses** – A list of tuples each containing a query, mc_type and result json.

**Return type** list[tuple(json, json)]

**get_updated_mc**(*mc_type*, *stmts*, *add_ns=False*, *edge_filter_func=None*)
Update the ModelChecker and graph with stmts for tests/queries.

**hash_response_list**(*response*)
Return a dictionary mapping a hash with a response in a response list.

**process_response**(*mc_type*, *result*)
Return a dictionary in which every key is a hash and value is a list of tuples. Each tuple contains a sentence describing either a step in a path (if it was found) or result code (if a path was not found) and a link leading to a webpage with more information about corresponding sentence.

**results_to_json**(*test_data=None*)
Put test results to json format.

**run_all_tests**(*filter_func=None*, *edge_filter_func=None*)
Run all applicable tests with all available ModelCheckers.

**run_tests_per_mc**(*mc_type*, *max_path_length*, *max_paths*, *filter_func=None*, *edge_filter_func=None*)
Run all applicable tests with one ModelChecker.

**save_assembled_statements**(*bucket='emmaa'*)
Upload assembled statements jsons to S3 bucket.

**upload_results**(*test_corpus='large_corpus_tests'*, *test_data=None*, *bucket='emmaa'*)
Upload results to s3 bucket.

**class** emmaa.model_tests.**RefinementTestConnector**
Bases: *emmaa.model_tests.TestConnector*

Determines applicability of a test to a model by checking if test entities or their refinements are in the model.

**static applicable**(*model*, *test*)
Return True of all test entities are in the set of model entities

**class** emmaa.model_tests.**ScopeTestConnector**
Bases: *emmaa.model_tests.TestConnector*

Determines applicability of a test to a model by overlap in scope.

**static applicable**(*model*, *test*)
Return True of all test entities are in the set of model entities

**class** emmaa.model_tests.**StatementCheckingTest**(*stmt*, *configs=None*)
Bases: *emmaa.model_tests.EmmaaTest*

Represent an EMMAA test condition that checks a PySB-assembled model against an INDRA Statement.

**check**(*model_checker*, *pysb_model*)
Use a model checker to check if a given model satisfies the test.

**get_entities**()
Return a list of entities that the test checks for.

**class** emmaa.model_tests.**TestConnector**
Bases: object

Determines if a given test is applicable to a given model.

**static applicable**(*model*, *test*)
Return True if the test is applicable to the given model.

**class** emmaa.model_tests.**TestManager**(*model_managers*, *tests*)
Bases: object

Manager to generate and run a set of tests on a set of models.

**Parameters**

- **model_managers** (*list[emmaa.model_tests.ModelManager]*) – A list of ModelManager objects
- **tests** (*list[emmaa.model_tests.EmmaaTest]*) – A list of EMMAA tests

**make_tests**(*test_connector*)
Generate a list of applicable tests for each model with a given test connector.

**Parameters** **test_connector** (emmaa.model_tests.TestConnector) – A TestConnector object to use for connecting models to tests.

**run_tests**(*filter_func=None*, *edge_filter_func=None*)
Run tests for a list of model-test pairs

emmaa.model_tests.**load_tests_from_s3**(*test_name*, *bucket='emmaa'*)
Load Emmaa Tests with the given name from S3.

**Parameters** **test_name** (*str*) – Looks for a test file in the emmaa bucket on S3 with key 'tests/{test_name}'.

**Returns** List of EmmaaTest objects loaded from S3.

**Return type** list of EmmaaTest

emmaa.model_tests.**model_to_tests**(*model_name*, *upload=True*, *bucket='emmaa'*)
Create StatementCheckingTests from model statements.

emmaa.model_tests.**run_model_tests_from_s3**(*model_name*, *test_corpus='large_corpus_tests'*, *upload_results=True*, *bucket='emmaa'*)
Run a given set of tests on a given model, both loaded from S3.

After loading both the model and the set of tests, model/test overlap is determined using a ScopeTestConnector and tests are run.

**Parameters**

- **model_name** (*str*) – Name of EmmaaModel to load from S3.
- **test_corpus** (*str*) – Name of the file containing tests on S3.
- **upload_results** (*Optional[bool]*) – Whether to upload test results to S3 in JSON format. Can be set to False when running tests. Default: True

**Returns** Instance of ModelManager containing the model data, list of applied tests and the test results.

**Return type** *emmaa.model_tests.ModelManager*

emmaa.model_tests.**save_tests_to_s3**(*tests*, *bucket*, *key*, *save_format='pkl'*)
Save tests in pkl, json or jsonl format.

## 3.4 Analyze model test results (emmaa.analyze_tests_results)

**class** emmaa.analyze_tests_results.**ModelRound**(*statements*, *date_str*, *paper_ids=None*, *paper_id_type='TRID'*, *emmaa_statements=None*)

Bases: *emmaa.analyze_tests_results.Round*

Analyzes the results of one model update round.

**Parameters**

- **statements** (*list[indra.statements.Statement]*) – A list of INDRA Statements used to assemble a model.
- **date_str** (*str*) – Time when ModelManager responsible for this round was created.
- **paper_ids** (*list (str)*) – A list of paper IDs used to get raw statements for this round.
- **paper_id_type** (*str*) – Type of paper ID used.

**stmts_by_papers**

A dictionary mapping the paper IDs to sets of hashes of assembled statements with evidences retrieved from these papers.

**Type** dict

**get_agent_distribution**()

Return a sorted list of tuples containing an agent name and a number of times this agent occured in statements of a model.

**get_all_raw_paper_ids**()

Return all paper IDs used in this round.

**get_assembled_stmts_by_paper**(*id_type='TRID'*)

Get a mapping of paper IDs (TRID or PII) to assembled statements.

**get_english_statements_by_hash**()

Return a dictionary mapping a statement and its English description.

**get_number_raw_papers**()

Return a total number of papers in this round.

**get_paper_titles_and_links**(*trids*)

Return a dictionary mapping paper IDs to their titles.

**get_papers_distribution**()

Return a sorted list of tuples containing a paper ID and a number of unique statements extracted from that paper.

**get_statement_types**()

Return a sorted list of tuples containing a statement type and a number of times a statement of this type occured in a model.

**get_statements_by_evidence**()

Return a sorted list of tuples containing a statement hash and a number of times this statement occured in a model.

**get_stmt_hashes**()

Return a list of hashes for all statements in a model.

**get_total_statements**()

Return a total number of statements in a model.

**class** emmaa.analyze_tests_results.**ModelStatsGenerator**(*model_name*, *latest_round=None*, *previous_round=None*, *previous_json_stats=None*, *bucket='emmaa'*)

Bases: *emmaa.analyze_tests_results.StatsGenerator*

Generates statistic for a given model update round.

**Parameters**

- **model_name** (*str*) – A name of a model the tests were run against.
- **latest_round** (emmaa.analyze_tests_results.ModelRound) – An instance of a ModelRound to generate statistics for. If not given, will be generated by loading model data from s3.
- **previous_round** (emmaa.analyze_tests_results.ModelRound) – A different instance of a ModelRound to find delta between two rounds. If not given, will be generated by loading model data from s3.
- **previous_json_stats** (*list[dict]*) – A JSON-formatted dictionary containing model statistics for previous update round.

**json_stats**

A JSON-formatted dictionary containing model statistics.

**Type** dict

**make_changes_over_time**()

Add changes to model over time to json_stats.

**make_curation_summary**()

Add latest curation summary to json_stats.

**make_model_delta**()

Add model delta between two latest model states to json_stats.

**make_model_summary**()

Add latest model state summary to json_stats.

**make_paper_delta**()

Add paper delta between two latest model states to json_stats.

**make_paper_summary**()

Add latest paper summary to json_stats.

**make_stats**()

Check if two latest model rounds were found and add statistics to json_stats dictionary. If both latest round and previous round were passed or found on s3, a dictionary will have three key-value pairs: model_summary, model_delta, and changes_over_time.

**class** emmaa.analyze_tests_results.**Round**(*date_str*)

Bases: object

Parent class for classes analyzing one round of something (model or tests).

**Parameters** **date_str** (*str*) – Time when ModelManager responsible for this round was created.

**function_mapping**

A dictionary of strings mapping a type of content to a tuple of functions necessary to find delta for this type of content. First function in a tuple gets a list of all hashes for a given content type, while the second returns an English description of a given content type for a single hash.

**Type** dict

**find_delta_hashes**(*other_round*, *content_type*, ***kwargs*)

Return a dictionary of changed hashes of a given content type. This method makes use of self.function_mapping dictionary.

**Parameters**

- **other_round** (emmaa.analyze_tests_results.TestRound) – A different instance of a TestRound
- **content_type** (*str*) – A type of the content to find delta. Accepted values: - statements - applied_tests - passed_tests - paths
- ****kwargs** (*dict*) – For some of content types, additional arguments must be provided sych as mc_type.

**Returns** **hashes** – A dictionary containing lists of added and removed hashes of a given content type between two test rounds.

**Return type** dict

**class** emmaa.analyze_tests_results.**StatsGenerator**(*model_name*, *latest_round=None*, *previous_round=None*, *previous_json_stats=None*, *bucket='emmaa'*)

Bases: object

Parent class for classes generating statistic for a given round of tests or model update.

**Parameters**

- **model_name** (*str*) – A name of a model the tests were run against.
- **latest_round** (ModelRound *or* TestRound *or* *None*) – An instance of a ModelRound or TestRound to generate statistics for. If not given, will be generated by loading json from s3.
- **previous_round** (ModelRound *or* TestRound *or* *None*) – A different instance of a ModelRound or TestRound to find delta between two rounds. If not given, will be generated by loading json from s3.
- **previous_json_stats** (*dict*) – A JSON-formatted dictionary containing model or test statistics for the previous round.

**json_stats**

A JSON-formatted dictionary containing model or test statistics.

**Type** dict

**make_changes_over_time**()

Add changes to model and tests over time to json_stats.

**class** emmaa.analyze_tests_results.**TestRound**(*json_results*, *date_str*)

Bases: *emmaa.analyze_tests_results.Round*

Analyzes the results of one test round.

**Parameters**

- **json_results** (*list[dict]*) – A list of JSON formatted dictionaries to store information about the test results. The first dictionary contains information about the model. Each consecutive dictionary contains information about a single test applied to the model and test results.

- **date_str** (*str*) – Time when ModelManager responsible for this round was created.

**mc_types_results**
A dictionary mapping a type of a ModelChecker to a list of test results generated by this ModelChecker

**Type** dict

**tests**
A list of INDRA Statements used to make EMMAA tests.

**Type** list[indra.statements.Statement]

**english_test_results**
A dictionary mapping a test hash and a list containing its English description, result in Pass/Fail/n_a form and either a path if it was found or a result code if it was not.

**Type** dict

**get_applied_test_hashes**()
Return a list of hashes for all applied tests.

**get_number_passed_tests**(*mc_type='pysb'*)
Return a number of all passed tests.

**get_passed_test_hashes**(*mc_type='pysb'*)
Return a list of hashes for passed tests.

**get_total_applied_tests**()
Return a number of all applied tests.

**passed_over_total**(*mc_type='pysb'*)
Return a ratio of passed over total tests.

**class** emmaa.analyze_tests_results.**TestStatsGenerator**(*model_name, test_corpus_str='large_corpus_tests', latest_round=None, previous_round=None, previous_json_stats=None, bucket='emmaa'*)

Bases: *emmaa.analyze_tests_results.StatsGenerator*

Generates statistic for a given test round.

**Parameters**

- **model_name** (*str*) – A name of a model the tests were run against.
- **test_corpus_str** (*str*) – A name of a test corpus the model was tested against.
- **latest_round** (emmaa.analyze_tests_results.TestRound) – An instance of a TestRound to generate statistics for. If not given, will be generated by loading test results from s3.
- **previous_round** (emmaa.analyze_tests_results.TestRound) – A different instance of a TestRound to find delta between two rounds. If not given, will be generated by loading test results from s3.
- **previous_json_stats** (*list[dict]*) – A JSON-formatted dictionary containing test statistics for previous test round.

**json_stats**
A JSON-formatted dictionary containing test statistics.

**Type** dict

**make_changes_over_time**()
Add changes to tests over time to json_stats.

**make_stats**()
Check if two latest test rounds were found and add statistics to json_stats dictionary. If both latest round and previous round were passed or found on s3, a dictionary will have three key-value pairs: test_round_summary, tests_delta, and changes_over_time.

**make_test_summary**()
Add latest test round summary to json_stats.

**make_tests_delta**()
Add tests delta between two latest test rounds to json_stats.

emmaa.analyze_tests_results.**generate_stats_on_s3**(*model_name*, *mode*, *test_corpus_str='large_corpus_tests'*, *upload_stats=True*, *bucket='emmaa'*)
Generate statistics for latest round of model update or tests.

**Parameters**

- **model_name** (*str*) – A name of EmmaaModel.
- **mode** (*str*) – Type of stats to generate (model or tests)
- **test_corpus_str** (*str*) – A name of a test corpus.
- **upload_stats** (*Optional[bool]*) – Whether to upload latest statistics about model and a test. Default: True

## 3.5 Query classes (emmaa.queries)

**class** emmaa.queries.**ComparativeInterventionProperty**
Bases: *emmaa.queries.Query*

**class** emmaa.queries.**DynamicProperty**(*entity*, *pattern_type*, *quant_value=None*, *quant_type='qualitative'*)
Bases: *emmaa.queries.Query*

This type of query requires dynamic simulation of the model to check whether the queried temporal pattern is satisfied.

**Parameters**

- **entity** (*indra.statements.Agent*) – An entity to simulate the model for.
- **pattern_type** (*str*) – Type of temporal pattern. Accepted values: 'always_value', 'no_change', 'eventual_value', 'sometime_value', 'sustained', 'transient'.
- **quant_value** (*str or float*) – Value of molecular quantity of entity of interest. Can be 'high' or 'low' or a specific number.
- **quant_type** (*str*) – Type of molecular quantity of entity of interest. Default: qualitative.

**get_temporal_pattern**(*time_limit=None*)
Return TemporalPattern object created with query properties.

**exception** emmaa.queries.**GroundingError**
Bases: Exception

**class** emmaa.queries.**OpenSearchQuery**(*entity*, *stmt_type*, *entity_role*, *terminal_ns=None*)

Bases: *emmaa.queries.Query*

This type of query requires doing an open ended breadth-first search to find paths satisfying the query.

**Parameters**

- **entity** (*indra.statements.Agent*) – An entity to simulate the model for.
- **stmt_type** (*str*) – Name of statement type.
- **entity_role** (*str*) – What role entity should play in statement (subject or object).
- **terminal_ns** (*list[str]*) – Force a path to terminate when any of the namespaces in this list are encountered and only yield paths that terminate at these namepsaces

**path_stmt**

An INDRA statement having its subject or object set to None to represent open search query.

**Type** indra.statements.Statement

**class** emmaa.queries.**PathProperty**(*path_stmt*, *entity_constraints=None*, *relationship_constraints=None*)

Bases: *emmaa.queries.Query*

This type of query requires finding a mechanistic causally consistent path that satisfies query statement.

**Parameters**

- **path_stmt** (*indra.statements.Statement*) – A path to look for in the model represented as INDRA statement.
- **entity_constraints** (*dict(list(indra.statements.Agent))*) – A dictionary containing lists of Agents to be included in or excluded from the path.
- **relationship_constraints** (*dict(list(str))*) – A dictionary containing lists of Statement types to include in or exclude from the path.

**get_entities**()

Return entities from the path statement and the inclusion list.

**class** emmaa.queries.**Query**

Bases: object

The parent class of all query types.

**class** emmaa.queries.**SimpleInterventionProperty**(*condition_entity*, *target_entity*, *direction*)

Bases: *emmaa.queries.Query*

This type of query requires dynamic simulation of the model to observe the behavior under perturbation.

**class** emmaa.queries.**StructuralProperty**

Bases: *emmaa.queries.Query*

emmaa.queries.**get_agent_from_gilda**(*ag_name*)

Return an INDRA Agent object by grounding its entity text with Gilda.

emmaa.queries.**get_agent_from_text**(*ag_text*)

Return an INDRA Agent object by grounding its entity text with either Gilda or TRIPS.

emmaa.queries.**get_agent_from_trips**(*ag_text*, *service_host='http://34.230.33.149:8002/cgi/'*)

Return an INDRA Agent object by grounding its entity text with TRIPS.

## 3.6 Process model queries (emmaa.answer_queries)

**class** emmaa.answer_queries.**QueryManager**(*db=None*, *model_managers=None*)

Bases: object

Manager to run queries and interact with the database.

**Parameters**

- **db** (*emmaa.db.EmmaaDatabaseManager*) – An instance of a database manager to use.
- **model_managers** (*list[emmaa.model_tests.ModelManager]*) – Optional list of ModelManagers to use for running queries. If not given, the methods will load ModelManager from S3 when needed.

**answer_immediate_query**(*user_email*, *user_id*, *query*, *model_names*, *subscribe*, *bucket='emmaa'*)

This method first tries to find saved result to the query in the database and if not found, runs ModelManager method to answer query.

**answer_registered_queries**(*model_name*, *bucket='emmaa'*)

Retrieve and asnwer registered queries

Retrieve queries registered on database for a given model, answer them, calculate delta between results and put results to a database.

**Parameters**

- **model_name** (*str*) – The name of the model
- **bucket** (*str*) – The bucket to save the results to

**get_registered_queries**(*user_email*, *query_type='path_property'*)

Get formatted results to queries registered by user.

**retrieve_results_from_hashes**(*query_hashes*, *query_type='path_property'*, *latest_order=1*)

Retrieve results from a db given a list of query-model hashes.

emmaa.answer_queries.**answer_queries_from_s3**(*model_name*, *db=None*, *bucket='emmaa'*)

Answer registered queries with model manager on s3.

**Parameters**

- **model_name** (*str*) – Name of EmmaaModel to answer queries for.
- **db** (*Optional[emmaa.db.manager.EmmaaDatabaseManager]*) – If given overrides the default primary database.

emmaa.answer_queries.**format_results**(*results*, *query_type='path_property'*)

Format db output to a standard json structure.

## 3.7 Priors (emmaa.priors)

This module contains classes to generate prior networks.

**class** emmaa.priors.**SearchTerm**(*type*, *name*, *db_refs*, *search_term*)

Bases: object

Represents a search term to be used in a model configuration.

**Parameters**

- **type** (*str*) – The type of search term, e.g. gene, bioprocess, other
- **name** (*str*) – The name of the search term, is equivalent to an Agent name
- **db_refs** (*dict*) – A dict of database references for the given term, is similar to an Agent db_refs dict
- **search_term** (*str*) – The actual search term to us for searching PubMed

**classmethod from_json**(*jd*)
Return a SearchTerm object from JSON.

**to_json**()
Return search term as JSON.

emmaa.priors.**get_drugs_for_gene**(*stmts*, *hgnc_id*)
Get list of drugs that target a gene

**Parameters**

- **stmts** (list of `indra.statements.Statement`) – List of INDRA statements with a drug as subject
- **hgnc_id** (*str*) – HGNC id for a gene

**Returns** **drugs_for_gene** – List of search terms for drugs targeting the input gene

**Return type** list of *emmaa.priors.SearchTerm*

### 3.7.1 Literature Prior (emmaa.priors.literature_prior)

This module implements the LiteraturePrior class which automates some of the steps involved in starting a model around a set of literature searches. Example:

```
lp = LiteraturePrior('some_disease', 'Some Disease',
                     'This is a self-updating model of Some Disease',
                     search_strings=['some disease'],
                     assembly_config_template='nf')
estmts = lp.get_statements()
model = lp.make_model(estmts, upload_to_s3=True)
```

emmaa.priors.literature_prior.**get_raw_statements_for_pmids**(*pmids*, *mode='all'*, *batch_size=100*)
Return EmmaaStatements based on extractions from given PMIDs.

**Parameters**

- **pmids** (*set or list of str*) – A set of PMIDs to find raw INDRA Statements for in the INDRA DB.
- **mode** (*'all' or 'distilled'*) – The 'distilled' mode makes sure that the "best", non-redundant set of raw statements are found across potentially redundant text contents and reader versions. The 'all' mode doesn't do such distillation but is significantly faster.
- **batch_size** (*Optional[int]*) – Determines how many PMIDs to fetch statements for in each iteration. Default: 100.

**Returns** A dict keyed by PMID with values INDRA Statements obtained from the given PMID.

**Return type** dict

emmaa.priors.literature_prior.**make_search_terms**(*search_strings*, *mesh_ids*)
Return EMMAA SearchTerms based on search strings and MeSH IDs.

**Parameters**

- **search_strings** (*list of str*) – A list of search strings e.g., "diabetes" to find papers in the literature.
- **mesh_ids** (*list of str*) – A list of MeSH IDs that are used to search the literature as headings associated with papers.

**Returns** A list of EMMAA SearchTerm objects constructed from the search strings and the MeSH IDs.

**Return type** list of emmmaa.prior.SearchTerm

### 3.7.2 TCGA Cancer Prior (emmaa.priors.cancer_prior)

**class** emmaa.priors.cancer_prior.**TcgaCancerPrior**(*tcga_study_prefix, sif_prior, diffusion_service=None, mutation_cache=None*)

Bases: object

Prior network generation using TCGA mutations for a given cancer type.

This class implements building a prior network using a generic underlying prior, and TCGA data for a specific cancer type. Mutations for the given cancer type are extracted from TCGA studies and heat diffusion from the corresponding nodes in the prior is used to identify a set of relevant nodes.

**static find_drugs_for_genes**(*node_list*)
Return list of drugs targeting gene nodes.

**get_mutated_genes**()
Return dict of gene mutation frequencies based on TCGA studies.

**get_relevant_nodes**(*pct_heat_threshold*)
Return a list of the relevant nodes in the prior.

Heat diffusion is applied to the prior network based on initial heat on nodes that are mutated according to patient statistics.

**load_sif_prior**(*fname, e50=20*)
Return a Graph based on a SIF file describing a prior.

**Parameters**

- **fname** (*str*) – Path to the SIF file.
- **e50** (*int*) – Parameter for converting evidence counts into weights over the interval [0, 1) according to hyperbolic function *weight = (count / (count + e50))*.

**make_prior**(*pct_heat_threshold=99*)
Run the prior node list generation and return relevant nodes.

**static search_terms_from_nodes**(*node_list*)
Build a list of Pubmed search terms from the nodes returned by make_prior.

### 3.7.3 Gene List Prior (emmaa.priors.gene_list_prior)

**class** emmaa.priors.gene_list_prior.**GeneListPrior**(*gene_list, name, human_readable_name*)

Bases: object

Class to manage the construction of a model from a list of genes.

**Parameters**

- **gene_list** (*list[str]*) – A list of HGNC gene symbols
- **name** (*str*) – The name of the model (all lower case, no spaces or special characters)
- **human_readable_name** (*str*) – The human readable name (display name) of the model

**make_config**()
Generate a configuration based on attributes.

**make_gene_statements**()
Generate Statements from the gene list.

**make_model**()
Make an EmmaaModel and upload it along with the config to S3.

**make_search_terms**(*drug_gene_stmts=None*)
Generate search terms from the gene list.

emmaa.priors.gene_list_prior.**agent_from_gene_name**(*gene_name*)
Return an Agent based on a gene name.

### 3.7.4 Reactome Prior (emmaa.priors.reactome_prior)

emmaa.priors.reactome_prior.**find_drugs_for_genes**(*search_terms, drug_gene_stmts=None*)
Return list of drugs targeting at least one gene from a list of genes

**Parameters** **search_terms** (list of *emmaa.priors.SearchTerm*) – List of search terms for genes

**Returns** **drug_terms** – List of search terms of drugs targeting at least one of the input genes

**Return type** list of *emmaa.priors.SearchTerm*

emmaa.priors.reactome_prior.**get_genes_contained_in_pathway**
Get all genes contained in a given pathway

**Parameters** **reactome_id** (*str*) – Reactome id for a pathway

**Returns** **genes** – List of uniprot ids for all unique genes contained in input pathway

**Return type** list of str

emmaa.priors.reactome_prior.**get_pathways_containing_gene**
"Get all ids for reactom pathways containing some form of an entity

**Parameters** **reactome_id** (*str*) – Reactome id for a gene

**Returns** **pathway_ids** – List of reactome ids for pathways containing the input gene

**Return type** list of str

emmaa.priors.reactome_prior.**make_prior_from_genes**(*gene_list*)
Return reactome prior based on a list of genes

**Parameters** **gene_list** (*list of str*) – List of HGNC symbols for genes

**Returns** **res** – List of search terms corresponding to all genes found in any reactome pathway containing one of the genes in the input gene list

**Return type** list of *emmaa.priors.SearchTerm*

`emmaa.priors.reactome_prior.`**`rx_id_from_up_id`**
Return the Reactome Stable IDs for a given Uniprot ID.

`emmaa.priors.reactome_prior.`**`up_id_from_rx_id`**
Get the Uniprot ID (referenceEntity) for a given Reactome Stable ID.

### 3.7.5 Querying Prior Statements (`emmaa.priors.prior_stmts`)

`emmaa.priors.prior_stmts.`**`get_stmts_for_gene`**(*gene*)
Return all existing Statements for a given gene from the DB.

**Parameters** **gene** (*str*) – The HGNC symbol of a gene to query.

**Returns** A list of INDRA Statements in which the given gene is involved.

**Return type** list[indra.statements.Statement]

`emmaa.priors.prior_stmts.`**`get_stmts_for_gene_list`**(*gene_list*, *other_entities*)
Return all Statements between genes in a given list.

**Parameters**

- **gene_list** (*list[str]*) – A list of HGNC symbols for genes to query.
- **other_entities** (*list[str]*) – A list of other entities to keep as part of the set of Statements.

**Returns** A list of INDRA Statements between the given list of genes and other entities specified.

**Return type** list[indra.statements.Statement]

## 3.8 Readers (`emmaa.readers`)

### 3.8.1 AWS reader (`emmaa.readers.aws_reader`)

`emmaa.readers.aws_reader.`**`read_pmid_search_terms`**(*pmid_search_terms*)
Return extracted EmmaaStatements given a PMID-search term dict.

**Parameters** **pmid_search_terms** (*dict*) – A dict representing a set of PMIDs pointing to search terms that produced them.

**Returns** A list of EmmaaStatements extracted from the given PMIDs.

**Return type** list[`emmaa.model.EmmaaStatement`]

`emmaa.readers.aws_reader.`**`read_pmids`**(*pmids*, *date*)
Return extracted INDRA Statements per PMID after running reading on AWS.

**Parameters**

- **pmids** (*list[str]*) – A list of PMIDs to read.
- **date** (*datetime*) – The date and time associated with the reading, typically the current time.

**Returns** A dict of PMIDs and the list of Statements extracted for the given PMID by reading.

**Return type** dict[str, list[indra.statements.Statement]

### 3.8.2 INDRA DB client reader (emmaa.readers.db_client_reader)

emmaa.readers.db_client_reader.**read_db_doi_search_terms**(*doi_search_terms*)

Return extracted EmmaaStatements from INDRA database given a DOI-search term dict.

**Parameters** **doi_search_terms** (*dict*) – A dict representing a set of DOIs pointing to search terms that produced them.

**Returns** A list of EmmaaStatements extracted from the given DOIs.

**Return type** list[emmaa.model.EmmaaStatement]

emmaa.readers.db_client_reader.**read_db_ids_search_terms**(*id_search_terms*, *id_type*)

Return extracted EmmaaStatements from INDRA database given an ID-search term dict.

**Parameters** **id_search_terms** (*dict*) – A dict representing a set of IDs pointing to search terms that produced them.

**Returns** A list of EmmaaStatements extracted from the given IDs.

**Return type** list[emmaa.model.EmmaaStatement]

emmaa.readers.db_client_reader.**read_db_pmid_search_terms**(*pmid_search_terms*)

Return extracted EmmaaStatements from INDRA database given a PMID-search term dict.

**Parameters** **pmid_search_terms** (*dict*) – A dict representing a set of PMIDs pointing to search terms that produced them.

**Returns** A list of EmmaaStatements extracted from the given PMIDs.

**Return type** list[emmaa.model.EmmaaStatement]

## 3.9 EMMAA's Database (emmaa.db)

### 3.9.1 The Database Schema (emmaa.db.schema)

**class** emmaa.db.schema.**User**(****kwargs*)

Bases: sqlalchemy.orm.decl_api.Base, emmaa.db.schema.EmmaaTable

A table containing users of EMMAA: User(_id_, email)

**Parameters**

- **id** (*int*) – (from indralab_auth_tools.src.models.User.id, primary key) A database-generated integer from the User table in indralab auth tools.
- **email** (*str*) – The email of the user (must be unique)

**class** emmaa.db.schema.**Query**(****kwargs*)

Bases: sqlalchemy.orm.decl_api.Base, emmaa.db.schema.EmmaaTable

Queries run on each model: Query(_hash_, model_id, json, qtype)

The hash column is a hash generated from the json and model_id columns that can be derived from the

**Parameters**

- **hash** (*big-int*) – (primary key) A 32 bit integer generated from the json and model_id.
- **model_id** (*str*) – (20 character) The short id/acronym for the given model.
- **json** (*json*) – A json dict containing the relevant parameters defining the query.

**class** emmaa.db.schema.**UserQuery**(***kwargs*)

Bases: sqlalchemy.orm.decl_api.Base, emmaa.db.schema.EmmaaTable

A table linking users to queries:

UserQuery(_id_, user_id, query_hash, date, subscription, count)

**Parameters**

- **id** (*int*) – (auto, primary key) A database-assigned integer id.
- **user_id** (*int*) – (foreign key -> User.id) The id of the user related to this query.
- **query_hash** (*big-int*) – (foreign key -> Query.hash) The hash of the query json, which can be directly generated.
- **date** (*datetime*) – (auto) The date that this entry was added to the database.
- **subscription** (*bool*) – Record whether the user has subscribed to see results of this model.
- **count** (*int*) – Record the number of times the user associated with user id has done this query

**class** emmaa.db.schema.**Result**(***kwargs*)

Bases: sqlalchemy.orm.decl_api.Base, emmaa.db.schema.EmmaaTable

Results of queries to models:

Result(_id_, query_hash, date, result_json, mc_type, all_result_hashes, delta)

**Parameters**

- **id** (*int*) – (auto, primary key) A database-assigned integer id.
- **query_hash** (*big-int*) – (foreign key -> Query.hash) The hash of the query json, which can be directly generated.
- **date** (*datetime*) – (auto) The date the result was entered into the database.
- **result_json** (*json*) – A json dict containing the results for the query.
- **mc_type** (*str*) – A name of a ModelChecker used to answer the query.

**class** emmaa.db.schema.**UserModel**(***kwargs*)

Bases: sqlalchemy.orm.decl_api.Base, emmaa.db.schema.EmmaaTable

A table linking users to models:

UserModel(_id_, user_id, model_id, date, subscription)

**Parameters**

- **id** (*int*) – (auto, primary key) A database-assigned integer id.
- **user_id** (*int*) – (foreign key -> User.id) The id of the user related to this query.
- **model_id** (*str*) – (20 character) The short id/acronym for the given model.
- **date** (*datetime*) – (auto) The date that this entry was added to the database.
- **subscription** (*bool*) – Record whether the user has subscribed to see results of this model.

### 3.9.2 Database Manager (emmaa.db.manager)

**class** emmaa.db.manager.**EmmaaDatabaseManager**(*host*, *label=None*)

Bases: object

A class used to manage sessions with EMMAA's database.

**add_user**(*user_id*, *email*)

Add a new user's email and id to Emmaa's User table.

**create_tables**(*tables=None*)

Create the tables from the EMMAA database

Optionally specify *tables* to be created. List may contain either table objects or the string names of the tables.

**drop_tables**(*tables=None*, *force=False*)

Drop the tables from the EMMAA database given in *tables*.

If *tables* is None, all tables will be dropped. Note that if *force* is False, a warning prompt will be raised to asking for confirmation, as this action will remove all data from that table.

**get_all_result_hashes**(*qhash*, *mc_type*)

Get a set of all result hashes for a given query and mc_type.

**get_model_users**(*model_id*)

Get all users who are subscribed to a given model.

**Parameters** **model_id** (*str*) – A standard name of a model to get users for.

**Returns** A list of email addresses corresponding to all users who are subscribed to this model.

**Return type** list[str]

**get_queries**(*model_id*)

Get queries that refer to the given model_id.

**Parameters** **model_id** (*str*) – The short, standard model ID.

**Returns** **queries** – A list of queries retrieved from the database.

**Return type** list[*emmaa.queries.Query*]

**get_results**(*user_email*, *latest_order=1*, *query_type=None*)

Get the results for which the user has registered.

**Parameters**

- **user_email** (*str*) – The email of a user.
- **latest_order** (*int*) – Which result in the order from the latest to get. Default: 1 ( latest).
- **query_type** (*str*) – Filter results to specific query type. Default: None (all query types will be returned).

**Returns** **results** – A list of tuples, each of the form: (model_id, query, mc_type, result_json, delta, date) representing the result of a query run on a model on a given date.

**Return type** list[tuple]

**get_subscribed_queries**(*email*)

Get a list of (query object, model id, query hash) for a user

**Parameters** **email** (*str*) – The email address to check subscribed queries for

**Returns**

**Return type** list(tuple(*emmaa.queries.Query*, str, query_hash))

**get_subscribed_users**()

Get all users who have subscriptions :returns: A list of email addresses corresponding to all users who have

any subscribed query

**Return type** list[str]

**get_user_models**(*email*)

Get all models a user is subscribed to.

**put_queries**(*user_email*, *user_id*, *query*, *model_ids*, *subscribe=True*)

Add queries to the database for a given user.

**Parameters**

- **user_email** (*str*) – the email of the user that entered the queries.
- **user_id** (*int*) – the user id of the user that entered the queries. Corresponds to the user id in the User table in indralab_auth_tools
- **query** (emmaa.queries.Query) – A query object containing all necessary information.
- **model_ids** (*list[str]*) – A list of the short, standard model IDs to which the user wishes to apply these queries.
- **subscribe** (*bool*) – True if the user wishes to subscribe to this query.

**put_results**(*model_id*, *query_results*)

Add new results for a set of queries tested on a model_id.

**Parameters**

- **model_id** (*str*) – The short, standard model ID.
- **query_results** (*list of tuples*) – A list of tuples of the form (query, mc_type, result_json), where the query is the query object run against the model, mc_type is the model type for the result, and the result_json is the json containing corresponding result.

**subscribe_to_model**(*user_email*, *user_id*, *model_id*)

Subsribe a user to model updates.

**Parameters**

- **user_email** (*str*) – the email of the user that entered the queries.
- **user_id** (*int*) – the user id of the user that entered the queries. Corresponds to the user id in the User table in indralab_auth_tools
- **model_id** (*str*) – Standard model ID to which the user wishes to subscribe.

**update_email_subscription**(*email*, *queries*, *models*, *subscribe*)

Update email subscriptions for user queries

NOTE: For now this method simply unsubscribes to the given queries but should in the future differentiated into recieving email notifications or not and subscribing to queries or not.

**Parameters**

- **email** (*str*) – The email assocaited with the query

- **queries** (*list(int)*) – A list of query hashes.
- **" list[str]** (*models*) – A list of models.
- **subscribe** (*bool*) – The subscription status for all matching query hashes

**Returns** Return True if the update was successful, False otherwise

**Return type** bool

**exception** emmaa.db.manager.**EmmaaDatabaseError**

Bases: Exception

## 3.10 AWS model update and testing pipeline (emmaa.aws_lambda_functions)

The AWS Lambda emmaa-update-pipeline definition.

This file contains the function that starts model update cycle. It must be placed on AWS Lambda, which can either be done manually (not recommended) or by running:

$ python update_lambda.py update_pipeline.py emmaa-update-pipeline

in this directory.

emmaa.aws_lambda_functions.update_pipeline.**lambda_handler**(*event*, *context*)

Invoke individual model update functions.

This function iterates through all models contained on S3 bucket and calls a different Lambda function to run model update for the models configured to be updated daily. It is expected that models have 'run_model_update' parameter in their config.json files.

This function is designed to be placed on AWS Lambda, taking the event and context arguments that are passed. Note that this function must always have the same parameters, even if any or all of them are unused, because we do not have control over what Lambda sends as parameters. Parameters are unused in this function.

Lambda is configured to automatically run this script every day.

See the top of the page for the Lambda update procedure.

**Parameters**

- **event** (*dict*) – A dictionary containing metadata regarding the triggering event.
- **context** (*object*) – This is an object containing potentially useful context provided by Lambda. See the documentation cited above for details.

**Returns** **ret** – A response returned by the latest call to emmaa-model-update function.

**Return type** dict

The AWS Lambda emmaa-model-update definition.

This file contains the function that starts model update cycle. It must be placed on AWS Lambda, which can either be done manually (not recommended) or by running:

$ python update_lambda.py model_updates.py emmaa-model-update

in this directory.

emmaa.aws_lambda_functions.model_updates.**lambda_handler**(*event*, *context*)

Create a batch job to update models on s3 and NDEx.

This function is designed to be placed on AWS Lambda, taking the event and context arguments that are passed. Note that this function must always have the same parameters, even if any or all of them are unused, because we do not have control over what Lambda sends as parameters. Event parameter is used to pass model_name argument.

This Lambda function is configured to be invoked by emmaa-update-pipeline Lambda function.

See the top of the page for the Lambda update procedure.

**Parameters**

- **event** (*dict*) – A dictionary containing metadata regarding the triggering event. In this case the dictionary contains model name.
- **context** (*object*) – This is an object containing potentially useful context provided by Lambda. See the documentation cited above for details.

**Returns** **ret** – A dict containing 'statusCode', with a valid HTTP status code, 'result', and 'job_id' to be returned to Lambda.

**Return type** dict

The AWS Lambda emmaa-mm-update definition.

This file contains the function that updates model manager object in S3. It must be placed on AWS Lambda, which can either be done manually (not recommended) or by running:

$ python update_lambda.py model_manager_update.py emmaa-mm-update

in this directory.

`emmaa.aws_lambda_functions.model_manager_update.`**`lambda_handler`**(*event*, *context*)

Create a batch job to update model manager on s3.

This function is designed to be placed on AWS Lambda, taking the event and context arguments that are passed. Note that this function must always have the same parameters, even if any or all of them are unused, because we do not have control over what Lambda sends as parameters. This Lambda function is configured to be triggered when the model is updated on S3.

See the top of the page for the Lambda update procedure.

**Parameters**

- **event** (*dict*) – A dictionary containing metadata regarding the triggering event. In this case, we are expecting 'Records', each of which contains a record of a file that was added (or changed) on s3.
- **context** (*object*) – This is an object containing potentially useful context provided by Lambda. See the documentation cited above for details.

**Returns** **ret** – A dict containing 'statusCode', with a valid HTTP status code, 'result', and 'job_id' to be returned to Lambda.

**Return type** dict

The AWS Lambda emmaa-after-update definition.

This file contains the function that will be run when Lambda is triggered. It must be placed on s3, which can either be done manually (not recommended) or by running:

$ python update_lambda.py after_update.py emmaa-after-update

in this directory.

`emmaa.aws_lambda_functions.after_update.lambda_handler`(*event*, *context*)

Submit model tests, model and test stats, and query batch jobs.

This function is designed to be placed on AWS Lambda, taking the event and context arguments that are passed. Note that this function must always have the same parameters, even if any or all of them are unused, because we do not have control over what Lambda sends as parameters. Event parameter is used here to pass which model manager was updated.

Lambda is configured to run this script when ModelManager object is updated.

**Parameters**

- **event** (*dict*) – A dictionary containing metadata regarding the triggering event. In this case, we are expecting 'Records', each of which contains a record of a file that was added (or changed) on s3.
- **context** (*object*) – This is an object containing potentially useful context provided by Lambda. See the documentation cited above for details.

**Returns** **ret** – A dict containing 'statusCode', with a valid HTTP status code, and any other data to be returned to Lambda.

**Return type** dict

The AWS Lambda emmaa-test-pipeline definition.

This file contains the function that will be run when Lambda is triggered. It must be placed on s3, which can either be done manually (not recommended) or by running:

$ python update_lambda.py test_pipeline.py emmaa-test-pipeline

in this directory.

`emmaa.aws_lambda_functions.test_pipeline.lambda_handler`(*event*, *context*)

Invoke individual test corpus functions.

This function is designed to be placed on lambda, taking the event and context arguments that are passed. Event parameter is used here to pass name of the model.

This Lambda function is configured to be invoked by emmaa-after-update Lambda function.

**Parameters**

- **event** (*dict*) – A dictionary containing metadata regarding the triggering event. In this case the dictionary contains 'model' key.
- **context** (*object*) – This is an object containing potentially useful context provided by Lambda. See the documentation cited above for details.

**Returns** **ret** – A response returned by the latest call to emmaa-model-test function.

**Return type** dict

The AWS Lambda emmaa-model-test definition.

This file contains the function that will be run when Lambda is triggered. It must be placed on s3, which can either be done manually (not recommended) or by running:

$ python update_lambda.py model_tests.py emmaa-model-test

in this directory.

`emmaa.aws_lambda_functions.model_tests.lambda_handler`(*event*, *context*)

Create a batch job to run model tests.

This function is designed to be placed on lambda, taking the event and context arguments that are passed. Event parameter is used here to pass names of the model and of the test corpus.

This Lambda function is configured to be invoked by emmaa-test-pipeline Lambda function.

**Parameters**

- **event** (*dict*) – A dictionary containing metadata regarding the triggering event. In this case the dictionary contains 'model' and 'tests' keys.
- **context** (*object*) – This is an object containing potentially useful context provided by Lambda. See the documentation cited above for details.

**Returns** **ret** – A dict containing 'statusCode', with a valid HTTP status code, and any other data to be returned to Lambda.

**Return type** dict

The AWS Lambda emmaa-test-stats definition.

This file contains the function that will be run when Lambda is triggered. It must be placed on s3, which can either be done manually (not recommended) or by running:

$ python update_lambda.py test_stats.py emmaa-test-stats

in this directory.

`emmaa.aws_lambda_functions.test_stats.`**`lambda_handler`**(*event*, *context*)

Create a batch job to generate model statistics.

This function is designed to be placed on lambda, taking the event and context arguments that are passed, and extracting the names of the uploaded (which includes changed) model or test definitions on s3. Lambda is configured to be triggered by any such changes, and will automatically run this script.

**Parameters**

- **event** (*dict*) – A dictionary containing metadata regarding the triggering event. In this case, we are expecting 'Records', each of which contains a record of a file that was added (or changed) on s3.
- **context** (*object*) – This is an object containing potentially useful context provided by Lambda. See the documentation cited above for details.

**Returns** **ret** – A dict containing 'statusCode', with a valid HTTP status code, and any other data to be returned to Lambda.

**Return type** dict

The AWS Lambda emmaa-model-stats definition.

This file contains the function that will be run when Lambda is triggered. It must be placed on s3, which can either be done manually (not recommended) or by running:

$ python update_lambda.py model_stats.py emmaa-model-stats

in this directory.

`emmaa.aws_lambda_functions.model_stats.`**`lambda_handler`**(*event*, *context*)

Create a batch job to generate model statistics.

This function is designed to be placed on lambda, taking the event and context arguments that are passed. Event parameter is used here to pass name of the model.

This Lambda function is configured to be invoked by emmaa-after-update Lambda function.

**Parameters**

- **event** (*dict*) – A dictionary containing metadata regarding the triggering event. In this case the dictionary contains 'model' key.
- **context** (*object*) – This is an object containing potentially useful context provided by Lambda. See the documentation cited above for details.

**Returns** **ret** – A dict containing 'statusCode', with a valid HTTP status code, and any other data to be returned to Lambda.

**Return type** dict

The AWS Lambda emmaa-queries definition.

This file contains the function that will be run when Lambda is triggered. It must be placed on s3, which can either be done manually (not recommended) or by running:

$ python update_lambda.py model_queries.py emmaa-queries

in this directory.

`emmaa.aws_lambda_functions.model_queries.lambda_handler`(*event*, *context*)

Create a batch job to run queries for model.

This function is designed to be placed on lambda, taking the event and context arguments that are passed. Event parameter is used here to pass name of the model.

This Lambda function is configured to be invoked by emmaa-after-update Lambda function.

**Parameters**

- **event** (*dict*) – A dictionary containing metadata regarding the triggering event. In this case the dictionary contains 'model' key.
- **context** (*object*) – This is an object containing potentially useful context provided by Lambda. See the documentation cited above for details.

**Returns** **ret** – A dict containing 'statusCode', with a valid HTTP status code, and any other data to be returned to Lambda.

**Return type** dict

The AWS Lambda emmaa-test-update-pipeline definition.

This file contains the function that starts model update cycle. It must be placed on AWS Lambda, which can either be done manually (not recommended) or by running:

$ python update_lambda.py test_update_pipeline.py emmaa-test-update-pipeline

in this directory.

`emmaa.aws_lambda_functions.test_update_pipeline.lambda_handler`(*event*, *context*)

Invoke individual model update functions.

This function iterates through all models contained on S3 bucket and calls a different Lambda function to turn the model into tests if the model is configured to do so. It is expected that models have 'make_tests' parameter in their config.json files.

This function is designed to be placed on AWS Lambda, taking the event and context arguments that are passed. Note that this function must always have the same parameters, even if any or all of them are unused, because we do not have control over what Lambda sends as parameters. Parameters are unused in this function.

Lambda is configured to automatically run this script every day.

See the top of the page for the Lambda update procedure.

**Parameters**

- **event** (*dict*) – A dictionary containing metadata regarding the triggering event.
- **context** (*object*) – This is an object containing potentially useful context provided by Lambda. See the documentation cited above for details.

**Returns** **ret** – A response returned by the latest call to emmaa-test-update function.

**Return type** dict

The AWS Lambda emmaa-test-update definition.

This file contains the function that updates tests created from model. It must be placed on AWS Lambda, which can either be done manually (not recommended) or by running:

$ python update_lambda.py test_update.py emmaa-test-update

in this directory.

`emmaa.aws_lambda_functions.test_update.`**`lambda_handler`**(*event*, *context*)

Create a batch job to update tests on s3.

This function is designed to be placed on AWS Lambda, taking the event and context arguments that are passed. Note that this function must always have the same parameters, even if any or all of them are unused, because we do not have control over what Lambda sends as parameters. Event parameter is used to pass model_name argument.

See the top of the page for the Lambda update procedure.

**Parameters**

- **event** (*dict*) – A dictionary containing metadata regarding the triggering event. In this case the dictionary contains model name.
- **context** (*object*) – This is an object containing potentially useful context provided by Lambda. See the documentation cited above for details.

**Returns** **ret** – A dict containing 'statusCode', with a valid HTTP status code, 'result', and 'job_id' to be returned to Lambda.

**Return type** dict

`emmaa.aws_lambda_functions.update_lambda.`**`upload_function`**(*script_name*, *function_name*)

Upload the lambda function by pushing a zip file to Lambda.

This function pre-supposes you are running from the same directory that contains the lambda script.

**Parameters**

- **script_name** (*str*) – Name of a script containing lambda function.
- **function_name** (*object*) – Name of a lambda function as specified on AWS Lambda.

## 3.11 xDD client

This modules provides an interface to query xDD content for figures and tables.

`emmaa.xdd.xdd_client.`**`get_document_figures`**(*paper_id*, *paper_id_type*)

Get figures and tables from a given paper.

**Parameters**

- **paper_id** (*str or int*) – ID of a paper.

- **paper_id_type** (*str*) – A name of a paper ID type (PMID, PMCID, DOI, TRID).

**Returns** **figures** – A list of tuples where each tuple is a figure title and bytes content.

**Return type** list[tuple]

emmaa.xdd.xdd_client.**get_document_objects**(*doi*)
Get a list of figure/table object dictionaries for a given DOI.

emmaa.xdd.xdd_client.**get_figures_from_objects**(*objects*, *paper_links=False*)
Get a list of paper links, figure titles and their content bytes from a list of object dictionaries (returned from query or document api).

emmaa.xdd.xdd_client.**get_figures_from_query**(*query*, *limit=None*)
Get figures and tables from a query.

**Parameters**

- **query** (*str*) – An entity name or comma-separated entity names to query for.
- **limit** (*int or None*) – A number of figures and tables to return.

**Returns** **figures** – A list of tuples where each tuple is a link to the paper, a figure title and bytes content.

**Return type** list[tuple]

emmaa.xdd.xdd_client.**send_document_search_request**(*doi*, *page*)
Send a request to get one page of results for a DOI.

emmaa.xdd.xdd_client.**send_query_search_request**(*query*, *page*)
Send a request to get one page of results for a query.

emmaa.xdd.xdd_client.**send_request**(*url*, *params*)
Send a request and handle potential errors.

## 3.12 EMMAA's Subscription Service (emmaa.subscription)

### 3.12.1 Notifications functions (emmaa.subscription.notifications)

**class** emmaa.subscription.notifications.**EmailHtmlBody**(*template_path*)
Bases: object

Parent class for email body.

**class** emmaa.subscription.notifications.**ModelDeltaEmailHtmlBody**(*template_path='email_unsub/model_ema*
Bases: *emmaa.subscription.notifications.EmailHtmlBody*

Email body for model updates.

**render**(*msg_dicts*, *unsub_link*)
Provided pregenerated msg_dicts render HTML to put in email body.

**Parameters**

- **msg_dicts** (*list[dict]*) – A list of dictionaries containing parts of messages to be added to email. Each dictionary has the following keys: 'url', 'start', 'delta_part', 'middle', 'message'.
- **unsub_link** (*str*) – A link to unsubscribe page.

**Returns** An html string rendered from the associated jinja2 template

**Return type** html

**class** emmaa.subscription.notifications.**QueryEmailHtmlBody**(*domain='emmaa.indra.bio', template_path='email_unsub/email_body.html'*)

Bases: *emmaa.subscription.notifications.EmailHtmlBody*

Email body for query notifications.

**render**(*static_query_deltas, open_query_deltas, dynamic_query_deltas, unsub_link*)

Provided the delta json objects, render HTML to put in email body.

**Parameters**

- **static_query_deltas** (*json*) – A list of lists that names which queries have updates. Expected structure: [(english_query, detailed_query_link, model, model_type)]
- **dynamic_query_deltas** (*list[*) – A list of lists that names which queries have updates. Expected structure: [(english_query, model, model_type)]
- **unsub_link** (*str*) – A link to unsubscribe page.

**Returns** An html string rendered from the associated jinja2 template

**Return type** html

emmaa.subscription.notifications.**get_all_update_messages**(*deltas, is_tweet=False*)

Get all messages for model deltas that can be further used in tweets and email notifications.

**Parameters**

- **deltas** (*dict*) – A dictionary containing deltas for a model and its test results returned by get_model_deltas function.
- **is_tweet** (*bool*) – Whether messages are generated for Twitter (used to determine the formatting of model types).

**Returns** **msg_dicts** – A list of individual message dictionaries that can be used for tweets or email notifications.

**Return type** list[dict]

emmaa.subscription.notifications.**get_model_deltas**(*model_name, test_corpora, date, bucket='emmaa'*)

Get deltas from model and test stats for further use in tweets and email notifications.

**Parameters**

- **model_name** (*str*) – A name of the model to get the updates for.
- **test_corpora** (*list[str]*) – A list of test corpora names to get the test updates for.
- **date** (*str*) – A date for which the updates should be generated.
- **bucket** (*str*) – A name of S3 bucket where the stats files are stored.

**Returns** **deltas** – A dictionary containing the deltas for the given model and test corpora.

**Return type** dict

emmaa.subscription.notifications.**get_user_query_delta**(*db, user_email, domain='emmaa.indra.bio'*)

Produce a report for all query results per user in a given format

**Parameters**

- **db** (*emmaa.db.EmmaaDatabaseManager*) – An instance of a database manager to use.

- **user_email** (*str*) – The email of the user for which to get the report for
- **domain** (*str*) – The domain name for the unsubscibe link in the html report. Default: "emmaa.indra.bio".

**Returns** A tuple with (str report, html report)

**Return type** tuple(str, html_str)

`emmaa.subscription.notifications.`**`make_html_report_per_user`**(*static_results_delta, open_results_delta, dynamic_results_delta, email, domain='emmaa.indra.bio'*)

Produce a report for all query results per user in an html file.

**Parameters**

- **static_results_delta** (*list*) – A list of tuples of query deltas for static queries. Each tuple has a format (english_query, link, model, mc_type)
- **open_results_delta** (*list*) – A list of tuples of query deltas for open queries. Each tuple has a format (english_query, link, model, mc_type)
- **dynamic_results_delta** (*list*) – A list of tuples of query deltas for dynamic queries. Each tuple has a format (english_query, link, model, mc_type)
- **email** (*str*) – The email of the user to get the results for.
- **domain** (*str*) – The domain name for the unsubscibe link in the report. Default: "emmaa.indra.bio".

**Returns** A string containing an html document

**Return type** str

`emmaa.subscription.notifications.`**`make_model_html_email`**(*msg_dicts, email, domain='emmaa.indra.bio'*)

Render html file for model notification email.

`emmaa.subscription.notifications.`**`make_reports_from_results`**(*new_results, domain='emmaa.indra.bio'*)

Make a report given latest results and queries the results are for.

**Parameters** **new_results** (*list[tuple]*) – Latest results as a list of tuples where each tuple has the format (model_name, query, mc_type, result_json, date, delta).

**Returns** **reports** – A list of reports on changes for each of the queries.

**Return type** list

`emmaa.subscription.notifications.`**`make_str_report_per_user`**(*static_results_delta, open_results_delta, dynamic_results_delta*)

Produce a report for all query results per user as a string.

**Parameters**

- **static_results_delta** (*list*) – A list of tuples of query deltas for static queries. Each tuple has a format (english_query, link, model, mc_type)
- **open_results_delta** (*list*) – A list of tuples of query deltas for open queries. Each tuple has a format (english_query, link, model, mc_type)

- **dynamic_results_delta** (*list*) – A list of tuples of query deltas for dynamic queries. Each tuple has a format (english_query, link, model, mc_type) (no link in dynamic_results_delta tuples).

**Returns** **msg** – A message about query deltas.

**Return type** str

emmaa.subscription.notifications.**model_update_notify**(*model_name*, *test_corpora*, *date*, *db*, *bucket='emmaa'*)

This function finds delta for a given model and sends updates via Twitter posts and email notifications.

**Parameters**

- **model_name** (*str*) – A name of EMMAA model.
- **test_corpora** (*list[str]*) – A list of test corpora names to get test stats.
- **date** (*str*) – A date for which to get stats for.
- **db** (*emmaa.db.EmmaaDatabaseManager*) – An instance of a database manager to use.
- **bucket** (*str*) – A name of S3 bucket where corresponding stats files are stored.

emmaa.subscription.notifications.**tweet_deltas**(*deltas*, *twitter_cred*)

Tweet the model updates.

**Parameters**

- **deltas** (*dict*) – A dictionary containing deltas for a model and its test results returned by get_model_deltas function.
- **twitter_cred** (*dict*) – A dictionary containing consumer_token, consumer_secret, access_token, and access_secret for a model Twitter account.

### 3.12.2 Email Service (emmaa.subscription.email_service)

emmaa.subscription.email_service.**close_to_quota_max**(*used_quota=0.95*, *region='us-east-1'*)

Check if the send quota is close to be exceeded

If the total quota for the 24h cycle is Q, the currently used quota is q and 'used_quota' is r, return True if q/Q > r, otherwise return False.

**Parameters**

- **used_quota** (*float*) – A float between 0 and 1.0. This number specifies the fraction of send quota currently used. Default: 0.95
- **region** (*str*) – A valid AWS region. The region to check the quota in. Default: us-east-1.

**Returns** True if the quota is close to be exceeded with respect to the provided ratio 'used'.

**Return type** bool

emmaa.subscription.email_service.**get_send_statistics**(*region='us-east-1'*)

Return the sending statistics, like bounce and complaint rates

See https://boto3.amazonaws.com/v1/documentation/api/latest/ reference/services/ses.html#SES.Client.get_send_statistics for more info

**Parameters** **region** (*Optional[str]*) – Specify AWS region

**Returns**

**Response syntax:**

```
{
  'SendDataPoints': [
    { 'Timestamp': datetime(2015, 1, 1), 'DeliveryAttempts': 123, 'Bounces': 123,
      'Complaints': 123, 'Rejects': 123
    },
  ]
}
```

**Return type** dict

`emmaa.subscription.email_service.`**`send_email`**(*sender*, *recipients*, *subject*, *body_text*, *body_html*, *source_arn=None*, *return_email=None*, *return_arn=None*, *region='us-east-1'*)

Wrapper function for the send_email method of the boto3 SES client

IMPORTANT: sending is limited to 14 emails per second.

See more at: https://boto3.amazonaws.com/v1/documentation/api/latest/reference + /services/ses.html#SES.Client.send_email https://docs.aws.amazon.com/ses/latest/APIReference/API_SendEmail.html and python example at https://docs.aws.amazon.com/ses/latest/DeveloperGuide/ + sending-authorization-delegate-sender-tasks-email.html

**Parameters**

- **sender** (*str*) – A valid email address to use in the Source field
- **recipients** (*iterable[str] or str*) – A valid email address or a list of valid email addresses. This will fill out the Recipients field.
- **subject** (*str*) – The email subject
- **body_text** (*str*) – The text body of the email
- **body_html** (*str*) – The html body of the email. Must be a valid html body (starting with <html>, ending with </html>).
- **source_arn** (*str*) – The source ARN of the sender. Should be of the format "arn:aws:ses:us-east-1:123456789012:identity/user@example.com" or "arn:aws:ses:us-east-1:123456789012:identity/example.com". Used only for sending authorization. It is the ARN of the identity that is associated with the sending authorization policy that permits the sender to send using the email address specified as the sender. Example: the owner of the domain "example.com" can send an email from any address using @example.com, as long as the associated source_arn is "arn:aws:ses:us-east-1:123456789012:identity/example.com"
- **return_email** (*str*) – The email to which complaints and bounces are sent. Can be the same as the sender.
- **return_arn** (*str*) – The return path ARN for the sender. This is the ARN associated with the return email. Can be the same as the source_arn if return email is the same as the sender.
- **region** (*str*) – AWS region to use for the SES client. Default: us-east-1

**Returns**

The API response object in the form of a dict is returned. The structure is:

```
>>> response = {                    'MessageId': 'EXAMPLE78603177f-
↪7a5433e7-8edb-42ae-af10' +                                    '-
↪f0181f34d6ee-000000',                    'ResponseMetadata': {
↪            '...': '...',                    },            }
```

**Return type** dict

### 3.12.3 Email Utilities (`emmaa.subscription.email_util`)

`emmaa.subscription.email_util.`**`generate_signature`**(*email*, *expire_str*, *digestmod=<built-in function openssl_sha256>*)

Return an HMAC signature based on email and expire_str

From documentation of HMAC in python: key is a bytes or bytearray object giving the secret key. If msg is present, the method call update(msg) is made. digestmod is the digest name, digest constructor or module for the HMAC object to use. It supports any name suitable to hashlib.new().

**Parameters**

- **email** (*str*) – A valid email address. Should not be URL encoded.
- **expire_str** (*str*) – A timestamp string in seconds
- **digestmod** (*str|digest constructor|module*) – digest name, digest constructor or module for the HMAC object to use. Default: hashlib.sha256

**Returns** A hexadecimal string representing the signature

**Return type** str

`emmaa.subscription.email_util.`**`generate_unsubscribe_link`**(*email*, *days=7*, *domain='emmaa.indra.bio'*)

Generate an unsubscribe link for the provided email address

Given an email address, generate an unsubscribe link using that email address. Optionally provide the number of days into the future the link should be valid until and the domain name. The domain name is expeceted to be of the format "some.domain.com". The appropriate path and prefixes will be added together with the query string. Example:

```
>>> generate_unsubscribe_link('user@email.com', domain='some.domain.com')
>>> 'https://some.domain.com/query/unsubscribe?email=user%40email.com' +
    '&expiration=1234567890&signature=1234567890abcdef'
```

**Parameters**

- **email** (*str*) – An email address.
- **days** (*int*) – The number of days into the future the link should be valid until. Default: 7.
- **domain** (*str*) – A domain name to prefix the query string with. Expected format is: "some.domain.com". Default: 'emmaa.indra.bio'

**Returns** An unsubscribe link for the provided email and (optionally) domain

**Return type** str

`emmaa.subscription.email_util.`**`generate_unsubscribe_qs`**(*email*, *days=7*)

Generate an unsubscribe query string for a url

**Parameters**

- **email** (*str*) – A valid email address
- **days** (*int*) – The number of days the query string should be valid. Default: 7.

**Returns** A query string of the format 'email=<urlenc email>&expiration=<timestamp>&signature=<sha256 hex>'

**Return type** str

`emmaa.subscription.email_util.`**`get_email_subscriptions`**(*email*)

Verifies which email subsciptions exist for the provided email

**Parameters** **email** (*str*) – The email to the check subscriptions for

**Returns**

**Return type** list(tuple(str, str, query_hash))

`emmaa.subscription.email_util.`**`register_email_unsubscribe`**(*email*, *queries*, *models*)

Executes an email unsubscribe request

`emmaa.subscription.email_util.`**`verify_email_signature`**(*signature*, *email*, *expiration*, *digestmod=<built-in function openssl_sha256>*)

Verify HMAC signature

## 3.13 Utilities (`emmaa.util`)

**exception** `emmaa.util.`**`NotAClassName`**

Bases: `Exception`

`emmaa.util.`**`does_exist`**(*bucket*, *prefix*, *extension=None*)

Check if the file with exact key or starting with prefix and/or with extension exist in a bucket.

`emmaa.util.`**`find_latest_emails`**(*email_type*, *time_delta=None*, *w_dt=False*)

Return a list of keys of the latest emails delivered to s3

**Parameters**

- **email_type** (*str*) – The email type to look for, e.g. 'feedback' if listing bounce and complaint emails sent to the ReturnPath address.
- **time_delta** (*datetime.timedelta*) – The timedelta to look backwards for listing emails.
- **w_dt** (*bool*) – If True, return a list of (key, datetime.datetime) tuples.

**Returns** A list of keys to the emails of the specified type. If w_dt is True, each item is a tuple of (key, datetime.datetime) of the LastModified date.

**Return type** list[Keys]

`emmaa.util.`**`find_latest_s3_file`**(*bucket*, *prefix*, *extension=None*)

Return the key of the file with latest date string on an S3 path

`emmaa.util.`**`find_latest_s3_files`**(*number_of_files*, *bucket*, *prefix*, *extension=None*)

Return the keys of the specified number of files with latest date strings on an S3 path sorted by date starting with the earliest one.

`emmaa.util.`**`find_nth_latest_s3_file`**(*n*, *bucket*, *prefix*, *extension=None*)

Return the key of the file with nth (0-indexed) latest date string on an S3 path

`emmaa.util.get_s3_client(unsigned=True)`

Return a boto3 S3 client with optional unsigned config.

**Parameters** **unsigned** (*Optional[bool]*) – If True, the client will be using unsigned mode in which public resources can be accessed without credentials. Default: True

**Returns** A client object to AWS S3.

**Return type** botocore.client.S3

`emmaa.util.make_date_str(date=None)`

Return a date string in a standardized format.

**Parameters** **date** (*Optional[datetime.datetime]*) – A date object to get the standardized string for. If not provided, utcnow() is used to construct the date. (Note: using UTC is important because this code may run in multiple contexts).

**Returns** The datetime string in a standardized format.

**Return type** str

`emmaa.util.sort_s3_files_by_date_str(bucket, prefix, extension=None)`

Return the list of keys of the files on an S3 path sorted by date starting with the most recent one.

`emmaa.util.sort_s3_files_by_last_mod(bucket, prefix, time_delta=None, extension=None, unsigned=True, reverse=False, w_dt=False)`

Return a list of s3 object keys sorted by their LastModified date on S3

**Parameters**

- **bucket** (*str*) – s3 bucket to look for keys in
- **prefix** (*str*) – The prefix to use for the s3 keys
- **time_delta** (*Optional[datetime.timedelta]*) – If used, should specify how far back the to look for files on s3. Default: None
- **extension** (*Optional[str]*) – If used, limit keys to those with the matching file extension. Default: None.
- **unsigned** (*bool*) – If True, use unsigned s3 client. Default: True.
- **reverse** (*bool*) – Reverse the sort order of the returned s3 files. Default: False.
- **w_dt** (*bool*) – If True, return list with datetime object along with key as tuple (key, datetime.datetime). Default: False.

**Returns** A list of s3 keys. If w_dt is True, each item is a tuple of (key, datetime.datetime) of the LastModified date.

**Return type** list

`emmaa.util.strip_out_date(keystring, date_format='datetime')`

Strips out datestring of selected date_format from a keystring

## 3.14 Functions for node and edge filtering (`emmaa.filter_functions`)

`emmaa.filter_functions.filter_chem_mesh_go(agent)`

Filter ungrounded agents and agents grounded to MESH, CHEBI, GO unless also grounded to HMDB.

`emmaa.filter_functions.`**`filter_to_internal_edges`**(*g*, *u*, *v*, **args*)

Return True if an edge is internal. NOTE it returns True if any of the statements associated with an edge is internal.

`emmaa.filter_functions.`**`register_filter`**(*filter_type*)

Decorator to register node or edge filter functions.

A node filter function should take an agent as an argument and return True if the agent is allowed to be in a path and False otherwise.

An edge filter function should take three (graph, source, target - for DiGraph) or three (graph, source, target, key - for MultiDiGraph) parameters and return True if the edge should be in the graph and False otherwise.

CHAPTER 4

# Configuring an EMMAA model

Each EmmaaModel has to be initiated with a config.json file. Config files can be generated manually or automatically with relevant methods in *Priors (emmaa.priors)* module (e.g. see *Literature Prior (emmaa.priors.literature_prior)* to start a model with default config from literature). This document describes the structure of the config.

## 4.1 First level fields of config.json

- ***name*** [str] A short name of a model.
  - Example: aml
- ***search_terms*** [list] A list of jsonified SearchTerms (see *emmaa.priors*) to search the literature for.
  - Example:

```
[{"type": "gene",
  "name": "PRKCA",
  "db_refs": {"HGNC": "9393", "UP": "P17252"},
  "search_term": "'PRKCA'"},
 {"type": "drug",
  "name": "SB 239063",
  "db_refs": {"HMS-LINCS": "10036",
              "PUBCHEM": "5166",
              "LINCS": "LSM-44951",
              "CHEBI": "CHEBI:91347"},
  "search_term": "'SB 239063'"}]
```

- ***human_readable_name*** [str] A human readable name of the model that will be displayed on the dashboard.
  - Example: Acute Myeloid Leukemia
- ***ndex*** [dict, optional] Configuration for NDEx network formatted as *{"network": <NDEx network ID>}*
  - Example:

```
{"network": "ef58f76d-f6a2-11e8-aaa6-0ac135e8bacf"}
```

- ***description*: str** Description of a model (will be displayed on EMMAA dashboard).
  - Example: A model of molecular mechanisms governing AML, focusing on frequently mutated genes, and the pathways in which they are involved.
- ***dev_only*** [bool, optional] Set to True if this model is still in development mode and should not be displayed on the main emmaa.indra.bio dashboard. Default: False.
- ***twitter*** [str, optional] If the model has Twitter account, this field should provide a key to retrieve Twitter secret keys stored on AWS SSM.
  - Example: covid19
- ***twitter_link*** [str, optional] URL to model's Twitter account if it exists.
  - Example: https://twitter.com/covid19_emmaa
- ***run_daily_update*** [bool] Whether the model should be updated with new literature daily.
- ***export_formats*** [list[str], optional] A list of formats the model can be exported to. Accepted values include: *indranet*, *pybel*, *sbml*, *kappa*, *kappa_im*, *kappa_cm*, *bngl*, *sbgn*, *pysb_flat*, *kappa_ui*. Note that *kappa_ui* option does not generate a separate export file but adds a link to Kappa interactive UI that uses model's kappa export (generated if *kappa* is in this list).
- ***assembly*** [dict or list[dict]] Configuration of model assembly represented as a dictionary where each key is a type of assembly (*main* for general purpose assembly steps and *dynamic* for additional steps to assemble a simulatable model) and values should contain corresponding jsonified steps to pass into the INDRA AssemblyPipeline class. Each step should have a *function* key and, if appropriate, *args* and *kwargs* keys. For more information on AssemblyPipeline, see https://indra.readthedocs.io/en/latest/modules/pipeline.html For backward compatibility, if a model has only one type of assembly (*main*), assembly configuration can be a list of steps instead of a dictionary with assembly types.
  - Example:

```
{"main": [
    {"function": "map_grounding",
     "kwargs": {"grounding_map": {
        "Viral replication": {"MESH": "D014779"},
        "viral replication cycle": {"MESH": "D014779"}}}},
    {"function": "run_preassembly",
     "kwargs": {"return_toplevel": false}},
    {"function": "filter_by_curation",
     "args": [{"function": "get_curations"},
              "any",
              ["correct", "act_vs_amt", "hypothesis"]],
     "kwargs": {"update_belief": true}}
    ],
 "dynamic": [
    {"function": "filter_by_type",
     "args": [{"stmt_type": "Complex"}],
     "kwargs": {"invert": true}},
    {"function": "filter_direct"},
    {"function": "filter_belief", "args": [0.95]}
    ]
}
```

- ***reading*** [dict, optional] Configuration of model update process. For more details see *Model update configuration*

- ***test*** [dict] Configuration of model testing. For more details see *Model testing configuration*
- ***query*** [dict, optional] Configuration of model queries. For more details see *Model queries configuration*
- ***make_tests*** [bool or dict, optional] It is possible to create tests from model assembled statements to test other models against them. If set to True, then tests will be created from all assembled statements. For details on filtering the statements to a specific subset, see *Making tests from model configuration*

## 4.2 Model update configuration

Model update configuration is the value mapped to the key *reading* in the model config. It defines the model update process. It can include the following fields:

- ***reader*** [list[str], optional] A list of readers to process the literature. Accepted elements are: *indra_db_pmid*, *indra_db_doi*, *elsevier_eidos*, *aws*. See *Readers (emmaa.readers)* for more information about readers. Default: ["indra_db_pmid"]
- ***literature_source*** [list[str], optional] A list of sources to search the literature. Accepted elements are: *pubmed*, *biorxiv*, *elsevier*. Default: ["pubmed"]. Note that literature sources should be provided in the same order as the readers to read them.
- ***cord19_update*** [dict, optional] COVID-19 specific configuration to update model from the CORD19 corpus. The dictionary should have the following fields:
  - ***metadata*** [dict] Metadata to pass to new EmmaaStatements.
  - ***date_limit*: int** Number of days to search back.
  - Example:

```
{"cord19_update": {
    "metadata": {
        "internal": true,
        "curated": false
        },
    "date_limit": 5
    }
}
```

- ***disease_map*** [dict, optional] A configuration to update a model from MINERVA Disease Map. It should have the following fields:
  - ***map_name*** [str] A name of a disease_map.
  - ***filenames*** [list[str] or str] A list of SIF filenames from the disease map to process or *all* to process all filenames.
  - ***metadata*** [dict] Metadata to pass to new EmmaaStatements.
  - Example:

```
{"disease_map": {
    "map_name": "covid19map",
    "filenames" : "all",
    "metadata": {
        "internal": true
        }
    }
}
```

- ***other_files*: list[dict]** A list of configurations to load statements from existing pickle files on S3. Each dictionary in the list should have the following fields:
  - ***bucket*** [str] A name of S3 bucket.
  - ***filename*** [str] A name of a pickle file.
  - ***metadata*** [str] Metadata to pass to new EmmaaStatements loaded from this file.
  - Example:

```
{"other_files": [
    {
        "bucket": "indra-covid19",
        "filename": "ctd_stmts.pkl",
        "metadata": {"internal": true, "curated": true}
    }
]
}
```

- ***filter*** [dict, optional] Configuration of a statement filter used for statistics generation (e.g. to not include external statements into statistics). The filter dictionary should have the following fields:
  - ***conditions*** [dict] Conditions represented as key-value pairs that statements' metadata can be compared to.
  - ***evid_policy*: str** Policy for checking statement's evidence objects. If "all", then the function returns True only if all of statement's evidence objects meet the conditions. If "any", the function returns True as long as at least one of statement's evidences meets the conditions.
  - Example:

```
{"filter": {
    "conditions": {"internal": true},
    "evid_policy": "any"
    }
}
```

## 4.3 Model testing configuration

Model testing configuration is the value mapped to the key *test* in the model config. It defines the model testing process. It can include the following fields:

- ***test_corpus*** [list[str]] A list of test corpora names that the model will be tested against daily.
  - Example : ["covid19_curated_tests", "covid19_mitre_tests"]
- ***default_test_corpus*** [str] The name of the test corpus that will be loaded by default on the model page on the EMMAA dashboard.
  - Example : "large_corpus_tests"
- ***mc_types*** [list[str]] A list of network types a model should be assembled into. For each of the model types, a ModelChecker instance will be created and used to find explanations to tests. Accepted elements are: *pysb*, *pybel*, *signed_graph*, *unsigned_graph*, *dynamic*.
- ***statement_checking*** [dict, optional] Maximum paths and maximum path length to limit test results. In the most general case the dictionary should have only two keys (*max_path_length* and *max_paths*) but it is also possible to set a custom configuration for one model type. In this case, a nested dictionary can be

added with model type as a key and a simple dictionary with the same two keys as a value. Default: {"max_path_length": 5, "max_paths": 1}.

  - Example (adding a custom config to a model type):

```
{"statement_checking": {
    "max_paths": 1,
    "max_path_length": 4,
    "pybel": {
        "max_paths": 1,
        "max_path_length": 10
        }
    }
}
```

- ***filters*** [dict] Configuration for applying semantic filters to the model checking process. It is represented as a dictionary mapping a test corpus name to a filter function name. The filter function should be defined in *Functions for node and edge filtering (emmaa.filter_functions)* and registered with *@register_filter('node')* decorator.

  - Example:

```
{"filters": {
    "covid19_mitre_tests" : "filter_chem_mesh_go"
    }
}
```

- ***edge_filters*** [dict] Configuration to apply edge filters to the model checking process. It is represented as a dictionary mapping a test corpus name to an edge filter function name. Filter function should be defined in *Functions for node and edge filtering (emmaa.filter_functions)* and registered with *@register_filter('edge')* decorator.

  - Example:

```
{"edge_filters": {
    "covid19_tests" : "filter_to_internal_edges"
    }
}
```

## 4.4 Model queries configuration

Configuration for model queries represented as a dictionary keyed by the type of query: *statement_checking* (source-target paths), *open_search* (up/down-stream paths), *dynamic* (temporal properties), and *intervention* (source-target dynamics). Configuration for *statement_checking* and *open_search* queries is similar to the model test *statement_checking* format. Same as in test config, it is possible to set different values for different model types.

Configuration for *dynamic* and *intervention* queries has different fields (all optional):

- ***use_kappa*** [bool] Determines the mode of the simulation. If True, uses *kappa*, otherwise, runs the ODE simulations. Default: False.
- ***time_limit*** [int] Number of seconds to run the simulation for. Default: 200000.
- ***num_times*** [int] Number of time points in the simulation plot. Default: 100.
- ***num_sim*** [int] Number of simulations to run. This should be only provided if *hypothesis_tester* is not set. Default: 2.

- ***hypothesis_tester*** [dict; currently only for *dynamic*, not *intervention*.] Configuration to test a hypothesis using random samples with adaptive size. If this is given, *num_sim* should not be provided. The *hypothesis_tester* dictionary should include the following keys: *alpha* (Type-I error limit, between 0 and 1), *beta* (Type-II error limit, between 0 and 1), *delta* (indifference parameter for interval around *prob* in both directions), *prob* (probability threshold for the hypothesis, between 0 and 1).

Having *dynamic* and *intervention* key in query config is required for a model to be listed as an option for model selection on temporal properties and source-target dynamics queries pages (for path-based queries all models will be listed).

- Example (all query types):

```
{"statement_checking": {
    "max_paths": 5,
    "max_path_length": 4,
    "pybel": {
        "max_paths": 10,
        "max_path_length": 10
        }
    },
 "open_search": {
    "max_paths": 50,
    "max_path_length": 2
    },
 "dynamic": {
    "use_kappa": true,
    "time_limit": 100,
    "num_times": 100,
    "hypothesis_tester": {"alpha": 0.1,
                          "beta": 0.1,
                          "delta": 0.05,
                          "prob": 0.8}
    },
 "intervention": {
    "use_kappa": true,
    "time_limit": 1000,
    "num_times": 100,
    "num_sim": 1
    },
}
```

## 4.5 Making tests from model configuration

Configuration to filter the statements before creating the tests (e.g. to make tests from literature derived statements and skip curated). It is the value mapped to the key *make_tests* in the model config (if you do not need to filter the statements and want to make tests from all assembled statements, it is enough to set *make_tests* to True). To filter statements, the *make_tests* should be set to dictionary with the key *filter* and the value should be another dictionary with the following fields:

- ***conditions*** [dict] Conditions represented as key-value pairs that statements' metadata can be compared to.
- ***evid_policy*: str** Policy for checking statement's evidence objects. If "all", then the function returns True only if all of statement's evidence objects meet the conditions. If "any", the function returns True as long as at least one of statement's evidences meets the conditions.

```
{"make_tests":
    {"filter": {
        "conditions": {"curated": false},
        "evid_policy": "any"
        }
    }
}
```

# CHAPTER 5

# ASKE Reports

This section contains reports on the EMMAA project as part of the DARPA Automating Scientific Knowledge Extraction (ASKE) program.

## 5.1 ASKE Month 5 Milestone Report: Lessons Learned

Here we summarize some of the high-level lessons we learned about large-scale machine-assisted model assembly and analysis over the course of developing EMMAA. Overall, we strongly believe that through an attempt to automate scientific modeling, we can gain substantial insight into the way human experts work with models of complex systems.

### 5.1.1 Automated model assembly: the challenge of defining scope and context

The initial development of EMMAA focused on deploying an automated model assembly pipeline to generate models specific to the various cancer types catalogued in the cancer genome atlas (TCGA). Collectively these models would constitute an "Ecosystem" of self-updating, context-specific models that could be used to answer mechanistic queries relevant to specific diseases. Context specificity was necessary because the answer to queries (e.g. "What is the effect of EGFR inhibition on cell growth?") differ depending on the specific gene expression pattern and mutation profile of a particular cancer type.

Our initial approach to enforce the context-specificity of automatically assembled models is described *here* and is centered on the genetics of specific cancer types. Frequently mutated genes in specific cancers were used as search terms to query Pubmed for publications which were then processed with machine reading tools and assembled into models along with information from curated databases.

Subsequent model testing highlighted a key shortcoming of this approach: tests of well-known biochemical pathways would fail in nearly all models because the limitations imposed on model scope (in the interest of context specificity) resulted in many key genes being omitted.

In an effort to expand models to incorporate key "backbone" genes while still retaining context specifity we then implemented two alternative approaches.

1. Run heat diffusion over our biological knowledge network to identify genes that were highly connected to the cancer-specific genes;

2. Query Reactome, a high quality database of biological pathways, for pathways containing the disease genes, and incorporate all genes from these pathways into the model.

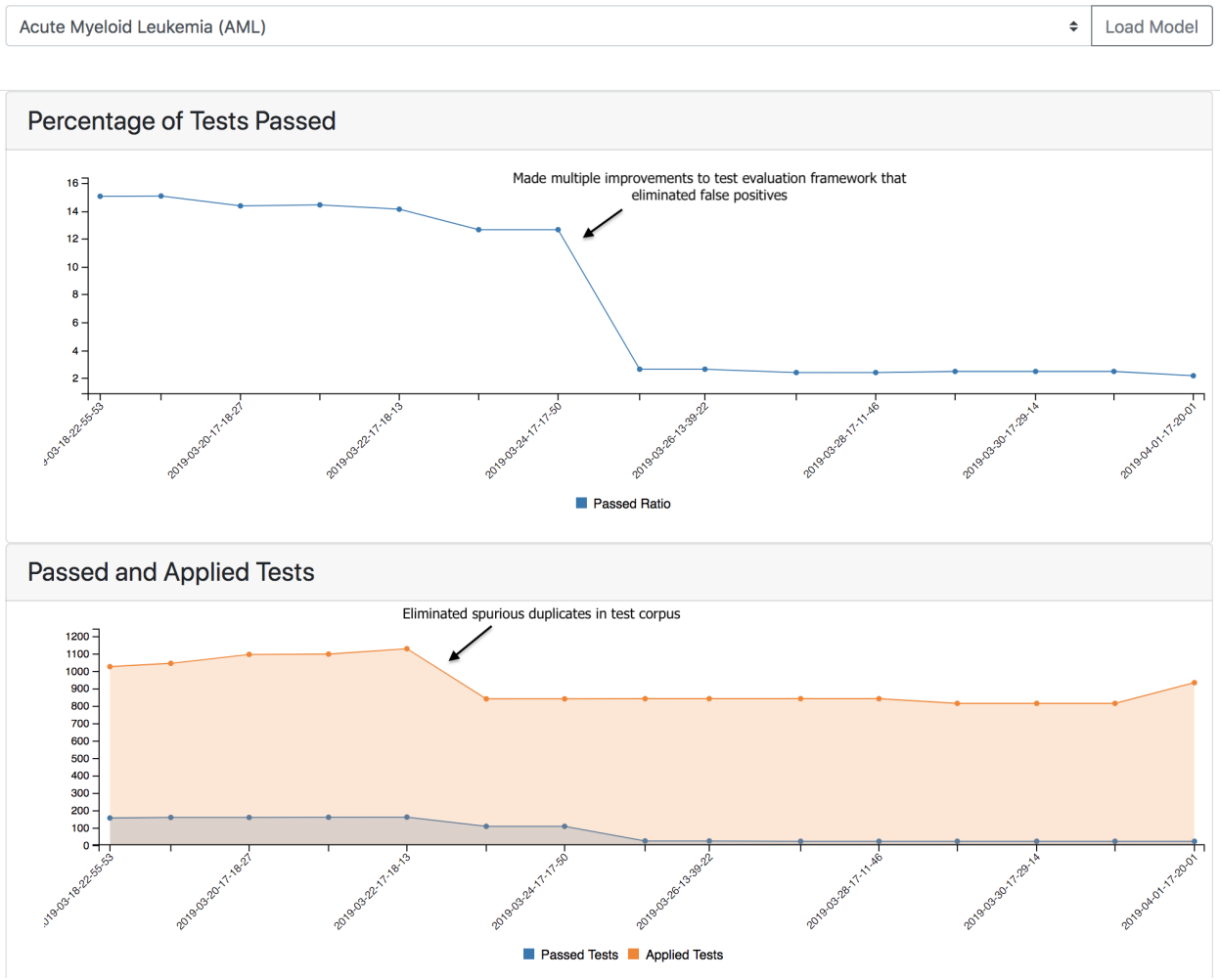
We found that the latter approach involving Reactome was more effective at eliminating mechanistic gaps than heat diffusion, which tended to highlight irrelevant genes based peculiarities of the knowledge network structure. However, even with the automated Reactome-based approach we found that models had a very low ratio of passing tests, and glaring mechanistic gaps: for example, the melanoma model passed only 4% of tests from the BEL Large Corpus, and omitted MAP2K1, a protein immediately downstream of (the frequently mutated gene) BRAF and a validated target in melanoma.

We therefore explored an alternative approach, in which models would be made specific to biochemical pathways rather than cancer types, a la the original Ras Machine. We found that the first iteration of this model had a much higher pass ratio of 34%, suggesting that models built and limited in scope in this way were more likely to have the internal integrity required for answering mechanistic queries.

Despite this improvement, the central problem of capturing model context remains: even if an automatically assembled model contains the genes relevant to a specific disease does not imply that it can answer a mechanistic query in a context specific way. For example, the Ras pathway is involved in many cancer types, not least in lung cancer and melanoma, yet the effects of intervening in the pathways differ between the two diseases. A key remaining challenge is to develop a system that can pull in the relevant data (e.g., gene expression, mutations) to contextualize structurally identical models, and make use of this data during analysis to reach context-specific conclusions.

### 5.1.2 Automated model analysis: benefits of automated model validation

With respect to model analysis, the first key lesson learned is how valuable the process of automated testing is for developing model assembly systems such as INDRA and EMMAA. By coupling large-scale automated reading and assembly with automated testing and analysis, the strengths and weaknesses of the reading/assembly machinery itself are clearly exposed. Over the course of monitoring daily updates to the disease models and browsing test results, we were able to identify bugs and other opportunities for improvement in a highly efficient and targeted way. The image below illustrates the effect of some of these improvements as they affected the number of applied and passed tests:




### 5.1.3 Test-driven modeling

A key observation that we have made during the development of EMMAA is of the value of automated model testing not only as a means of post-hoc model validation, but also to support *test-driven modeling*. That is, the *construction* (in part manual) of scientific models based on a corpus of qualitative experimental constraints. This is by analogy with test-driven development in software engineering in which the tests are written first, and program features are only added to satisfy the tests.

During this reporting period, we have explored test-driven modeling by manually building a model of a core subset of the Ras signaling pathway. The model is built using natural language via INDRA and TRIPS as described here; the automated assembly of the natural language sentences yields a model with semantic annotations enabling subsequent testing and analysis. The model is exposed in the EMMAA dashboard as the "Ras Model". The initial model consisted of a set of roughly 60 natural language sentences and was roughly doubled in size through an iterative process of expansion and refinement that was driven by model testing.

We have found that the test-driven modeling approach has a number of advantages for the construction of scientific models. First, the approach to scientific modeling in many fields is to use a formal model to encode a specific hypothesis about a particular phenomenon. These "fit-to-purpose" models are useful tools for answering specific scientific questions but they are rarely reusable and are biased toward a particular explanation. With test-driven modeling, the growth of the model is empirically driven by the observations that match the scope of the model, independent of any specific problem. In extending the Ras model to satisfy tests from the BEL Large Corpus, we repeatedly found it nec-

essary to add in underappreciated or noncanonical mechanisms. For example, it is well known that EGFR activation leads to the phosphorylation and activation of SRC; but it is also the case that SRC phosphorylates and potentiates the activation of EGFR. Similarly, AKT1 both phosphorylates and is phosphorylated by MTOR. In typical practice, a modeler would not incorporate all of these influences unless it was their specific intention to investigate crosstalk, feedback, or other aspects of the overall mechanism that deviate from a simple linear pathway. The process of test-driven modeling brought to the forefront how common these processes are.

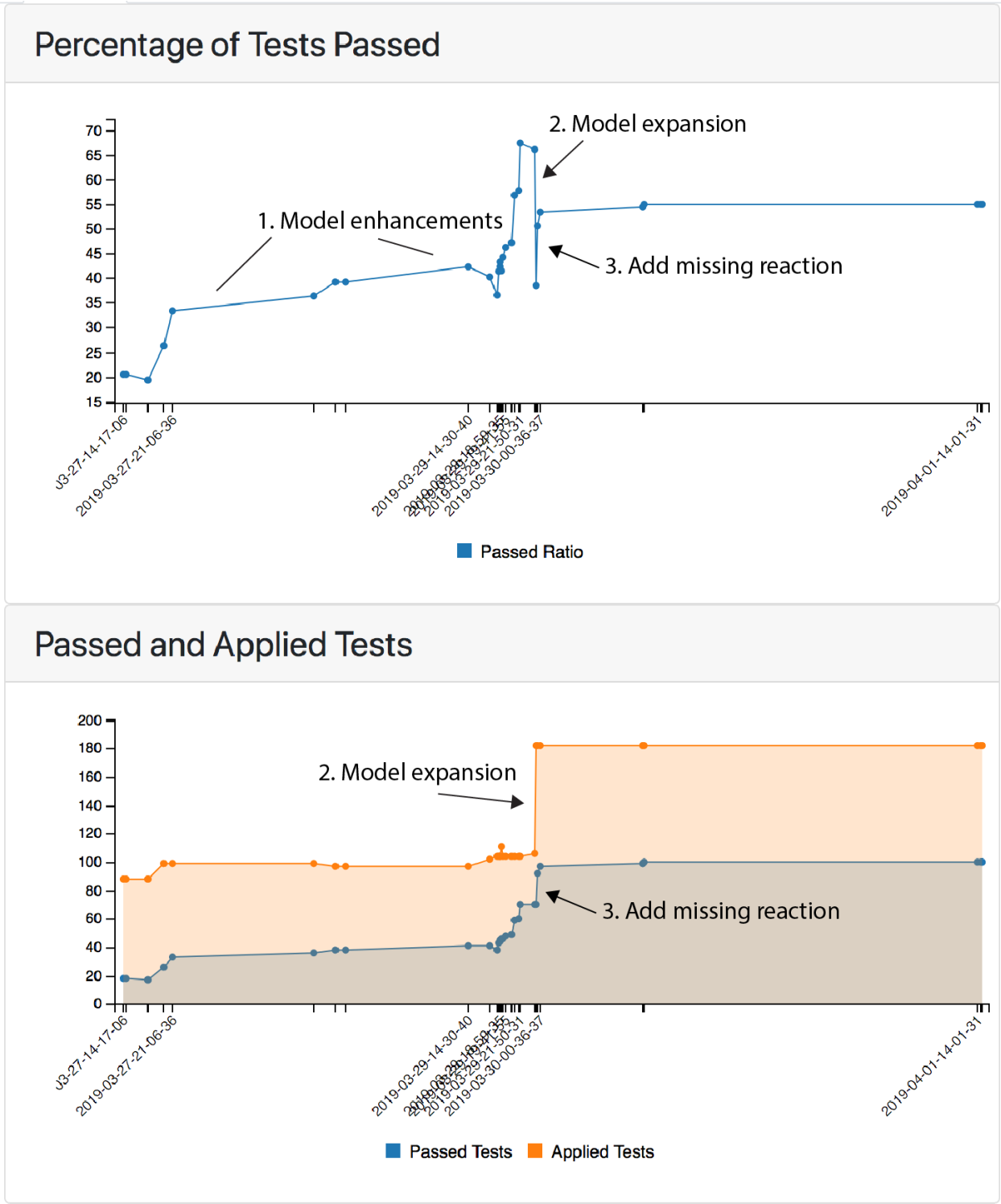
Second, just as in software development, test-driven modeling helps the modeler avoid decorating a model with details that are not essential to improving overall performance. This helps to avoid modeling quagmires in which a modeler attempts to encode everything known about a process in maximum detail. The existence of a set of tests, and the iterative development process that EMMAA enables (serving here as a tool for continuous integration of models), dramatically improves the efficiency of building high quality, reusable models.

Third, test-driven modeling helps build insight into how a model works, as well as highlighting serendipitous and potentially unexpected implications of particular mechanisms. During the test-driven development of the Ras Model, there were several instances where adding a small extension to the model to address an issue that appeared to be local to the two proteins resulted in several additional tests passing, that involved long-range causal influences. For example, fixing a reaction involving MTOR and PPP2CA resulted in three tests passing, each highlighting the negative feedback from MTOR back to upstream IGF1R signaling via IRS-1.

## New Passed Tests

| Test | Top Path |
|---|---|
| **New passed tests for PyBEL model.** | |
| BMP2 activates CASP9. | BMP2 → PTEN → BAX → CASP9 |
| BMP2 increases the amount of CCND3. | BMP2 → PTEN → PIK3CA → CCND3 |
| BMP2 activates MEK. | BMP2 → PTEN → KRAS → MEK |
| BMP2 inhibits STAT3. | BMP2 → PTEN → EGFR → STAT3 |
| **New passed tests for Signed Graph model.** | |
| BMP2 activates CASP9. | BMP2 → PTEN → BAX → CASP9 |
| BMP2 increases the amount of CCND3. | BMP2 → PTEN → PIK3CA → CCND3 |
| BMP2 activates MEK. | BMP2 → PTEN ⊣ EGFR ⊣ MEK |
| BMP2 inhibits cell cycle. | BMP2 → PTEN → RB1 ⊣ cell cycle |
| BMP2 activates cell differentiation. | BMP2 → PTEN → cell differentiation |
| BMP2 inhibits STAT3. | BMP2 → PTEN ⊣ EGFR → STAT3 |
| **New passed tests for Unsigned Graph model.** | |
| BMP2 activates CASP9. | BMP2 → PTEN → BAX → CASP9 |

The screenshot of the EMMAA dashboard test results page for the curated Ras Model, shown below, highlights the iterative process of test-driven model refinement and expansion.




The bottom plot shows the total number of applied tests over time, along with the number of passing tests; the top plot tracks changes in the percentage of passing tests. The initial process of model refinement is shown by (1), in which the initial model was subject to testing and then progressively refined over time. During this process the pass ratio grew from roughly 20% to 67%. At this point, the model was expanded to include the well studied signaling

proteins EGF and EGFR. This nearly doubled the number of applied tests (2, bottom plot), but since relatively few of these new tests passed, the pass ratio dropped to ~35%. Importantly, these new tests were applied *automatically* by EMMAA as a consequence of the expansion in model scope. Inspection of the model highlighted the fact that EGFR was disconnected from many of its downstream effectors; addition of only a single statement (connecting EGFR to SOS1, which was already in the model for its role downstream of IGF1R) led to a large number of the new tests passing, boosting the pass ratio back to over 50% (3, both plots).

### 5.1.4 Exploiting the bidirectional relationship between models and tests

During the development of EMMAA we have come to appreciate the benefits of treating the information flow between models and tests as symmetric and bidirectional.

For example, manually curated tests can be used to validate automatically assembled models, or the other way around: curated models validating automatically extracted observations. In our initial work, we focused on the application of curated experimental observations (from the BEL large corpus) to automatically assembled mechanistic models. We described above how applying these tests to the Ras Machine model helped us to identify issues in our automatic model assembly pipeline that had been latent for years. We now also see the value in automatically collecting tests and using high-quality curated models to evaluate the plausibility of the test observations themselves. For example, in the development of the Ras Model, we found that a surprising proportion (over 15%) of the tests in the BEL Large Corpus were incorrectly curated. These *test errors* were inadvertently highlighted when the *model* failed to pass them. We imagine that observations derived from a noisy source (such as machine reading) could be subjected to checking by one or more high-quality models, with the model establishing the likelihood that a finding resulted from a machine reading error. It is also possible to imagine that in fields where models are mature, new scientific findings could be automatically subjected to model-driven evaluation, highlighting the ways in which they either support or contradict established models.

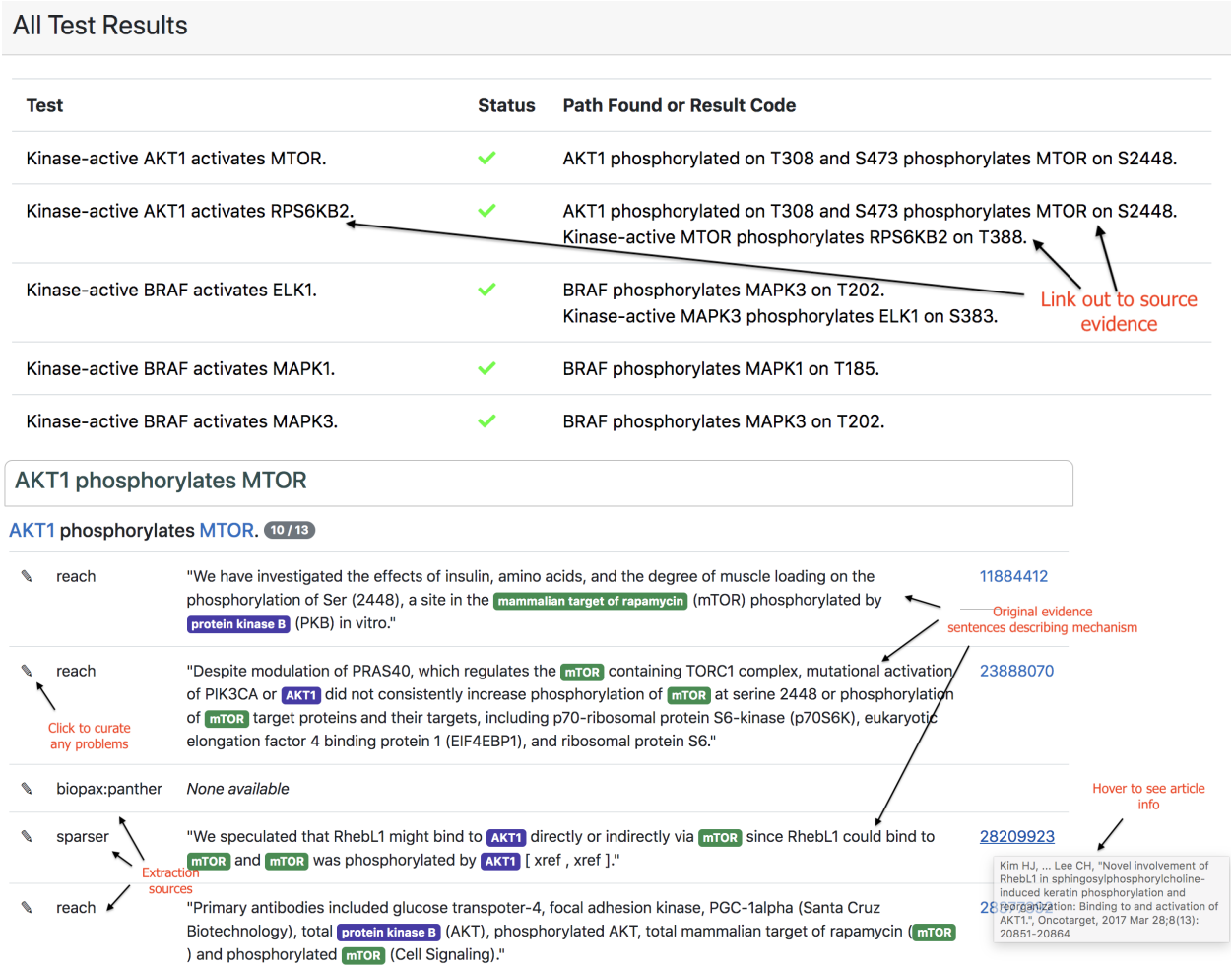
## 5.2 ASKE Month 6 Milestone Report

### 5.2.1 Making model analysis and model content fully auditable

When browsing the results of model tests, it is often of interest to inspect the specific provenance of each modeled mechanisms that contributed to the result. EMMAA models are built automatically from primary knowledge sources (databases and literature), and model components are annotated such that given the result, we can link back to the original sources.

Links to browse evidence are available in all of the following contexts:

- New statements added to the model
- Most supported statements in the model
- New tests applicable to the model
- Passed/failed tests and the mechanisms constituting paths by which tests passed




### 5.2.2 Including new information based on relevance

EMMAA models self-update by searching relevant litearture each day and adding mechanisms described in new publications. However, event publications that are relevant often contain pieces of information that aren't directly relevant for the model. We therefore created a relevance filter which can take one of several policies and determine if a new statement is *relevant* to the given model or not. The strictest policy is called *prior_all* which only considers statements in which all participants are prior search terms of interest for the model as relevant. A less strict policy, *prior_one* requires that at least one participant of a statement is a prior search term for the model. Currently, EMMAA models are running with the *prior_one* policy.

### 5.2.3 Coarse-grained model checking of EMMAA models with directed graphs

To determine whether a model can satisfy a particular test, EMMAA currently assembles sets of INDRA Statements into mechanistic PySB/Kappa models. The INDRA ModelChecker is then used to determine whether there is a causal path in the Kappa influence map linking the subject and object of the test with the appropriate causal constraints. These constraints include the polarity of the path, the detailed attributes of the subject and object (for example, a particular modified form of the object protein), and the type of regulation (e.g., regulation of activity vs. regulation of amount). Because the assembled PySB/Kappa models make maximum use of available mechanistic information, this approach to model checking yields results with high precision, in that the existence of a path indicates that the strict semantics of the test are satisfied.

The high precision of this approach comes at the expense of recall and robustness, in that tests may *not* pass due to subtle aspects of the test or model statements. For example, if a machine reading system incorrectly extracts a positive regulation statement linking genes *A* and *B* as a regulation of amount rather than a regulation of activity, this can lead to the test "A activates B" failing and yielding no paths.

To help scientists using EMMAA to generate scientific insight, it would be ideal for models to be verified against tests with *different degrees of causal constraints.* If a model fails to satisfy a test using the high-precision approach, the scientist user could also inspect causal paths produced by model assembly and checking procedures with a more generous interpretation of causality.

A key advantage of using INDRA as the model assembly engine within EMMAA is that a single knowledge representation (INDRA Statements) can be used to assemble multiple types of causal models. In the context of EMMAA, INDRA can be used to assemble at least four different types of models, listed in increasing order of causal precision:

- Directed networks
- Signed directed networks
- Boolean networks
- Biological Expression Language (BEL) networks
- PySB model/Kappa influence maps

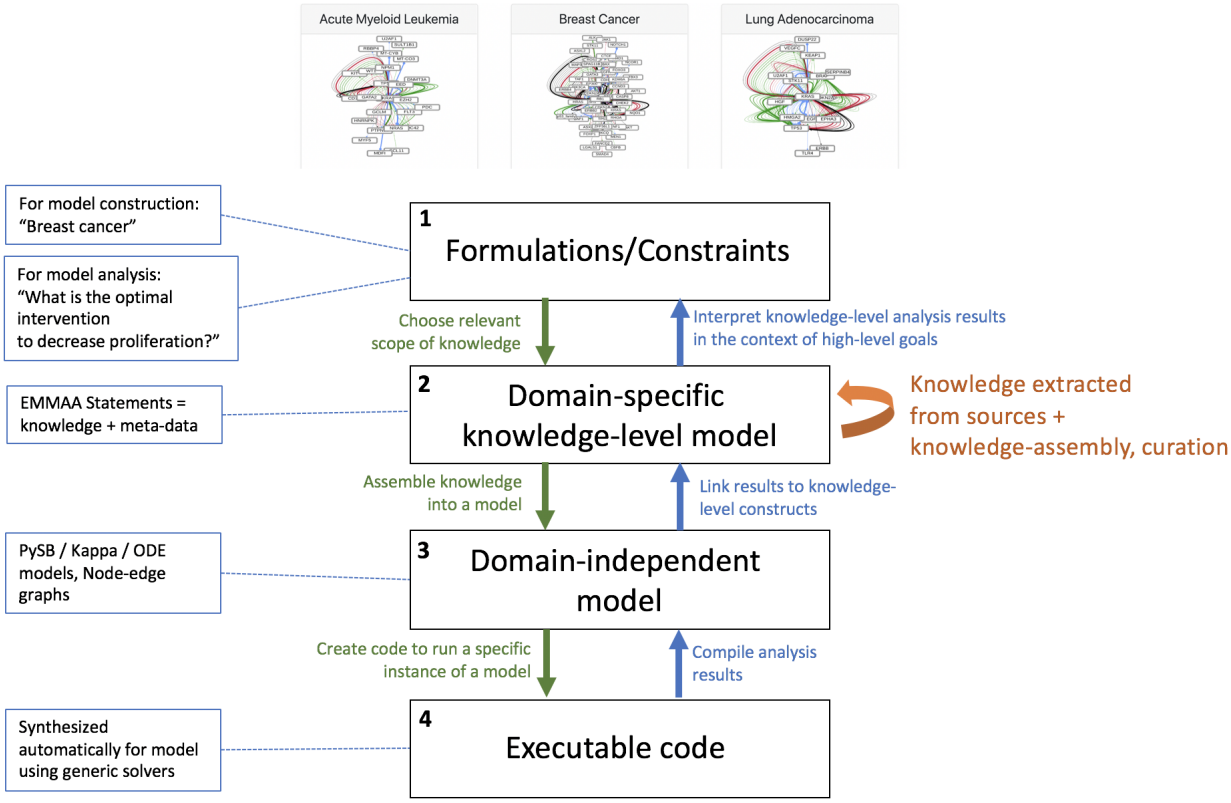
During this reporting period, we investigated the use of the most coarse-grained of these representations, directed networks, to check EMMAA models against tests. Code and results are available in an iPython notebook accompanying this report available on Github here. Using the most recent model and test results from the EMMAA Ras Machine 2.0, we built a simple directed graph among agents using *networkx* and checked for paths between pairs of genes in the applied tests.

We found that, as expected, many more tests passed in the directed graph model (509 tests, 58.7%) than the detailed PySB/Kappa model (165 tests, 19.0%). All tests passed by the PySB/Kappa model also passed in the directed graph model, indicating that the latter is a strict subset of the former. Roughly half (52%) of the tests that *failed* in the PySB/Kappa model yielded paths in the directed graph. Inspection of the discrepancies highlighted some characteristics of these types of tests (see iPython notebook.) A key point is that the proportion of tests passed by the directed graph model represent an *upper bound* of the mechanistic coverage of the model that is independent of the particular modeling formalism involved. While many of the paths found in the directed graph do not satisfy the strictest interpretation of the tests, they are nevertheless useful for a human scientist to better understand relevant processes contained in the model and to generate hypotheses.

In the upcoming reporting period we aim to extend this approach further by using EMMAA to assemble multiple types of models at different levels of causal resolution. A scientist will then be able to explore a range of explanations for a given observation depending on the precision-recall tradeoffs of their use case.

## 5.3 ASKE Month 7 Milestone Report

### 5.3.1 Repositioning EMMAA within the ASKE framework of modeling layers




Based on discussions at the ASKE 6-month PI meeting we have been reformulating how our approach in EMMAA relates to the three proposed modeling representation levels. One outcome of the meeting was an emerging consensus that the middle "model" level represents domain-independent model representations that are not yet executable. Examples discussed included linear regression models, polynomial functions, ordinary differential equations, etc. Prior to this discussion we had considered this layer, representing classes of mathematical models, to be the "bottom"; however, we now recognize that a model at this level is not yet executable because it must first be coupled to a particular simulation or inference procedure.

With this in mind, we feel that our approach requires the definition of an additional layer, sitting between the topmost (level 1, "formulations/constraints") and the mathematical modeling layer (now level 3). This layer corresponds to networks of EMMAA/INDRA Statements: representations of a particular subset of domain *knowledge*. A knowledge network at this layer may be formulated based on requirements/constraints specified in level 1 (e.g., "a knowledge model of breast cancer", or "all signaling pathways with a bowtie architecture"). In turn, this knowledge network can be used to generate different analytical/mathematical models at level 3 (boolean networks, rule-based models, analytical/mathematical model ODEs, etc.).

One focus of the discussion during the PI meeting was on the potential for integration of ASKE modeling frameworks via the domain-independent level 3. Integration at this level would allow tools for model analysis, simulation, expansion, etc. to be reused between teams. At this layer domain-specific considerations may still apply but they will have been converted into syntactic constraints expressed in the language of the particular modeling formalism. One example relevant to biology is the formulation of ODE models: while in a domain-independent sense the class of all ordinary differential equation models is quite large, biological models typically make use of a highly restricted subset of mathematical functions. In a "mass-action" reaction model, for example, the right hand side function consists strictly of a linear combination of products of the concentration variables. This (semantic) biochemical constraint

could be expressed in the (syntactic) language of mathematical functions to allow the application of tools for model expansion, simulation, etc.

Since the PI meeting we have also concluded that important model inference and transformation procedures can occur at layers other than level 3, and that these operations can occur *within*, not just across layers. For example, in INDRA there are a set of related procedures that we collectively refer to as "knowledge assembly" or "pre-assembly": identifying subsumption relationships, inferring and applying belief scores, identifying statement relationships, etc. Both the information considered in these operations, and the operations themselves explicitly make use of domain-specific knowledge, and all take INDRA Statements as input and produce INDRA Statements as output. These steps are referred to as "pre-assembly" to differentiate them from the step of *assembly*, which denotes the transformation of knowledge-level information (level 2) from model-level information (level 3).

### 5.3.2 Use cases for the EMMAA system (and ASKE systems in general)

#### Push Science

In 2015, Paul Cohen defined "push scholarship" as: "[. . . ] instead of pulling results into our heads, we push results into machine-maintained big mechanisms, where they can be examined by anyone. This could change science profoundly."

ASKE systems have the potential to go beyond this ambitious goal by:

- actively searching for new discoveries and data,
- autonomously updating a set of models by integrating new discoveries,
- designing model analysis experiments to understand the effect of this new knowledge
- reporting the effect of new discoveries on scientific questions relevant to the user

In other words, novel, relevant implications of discoveries, as soon as they appear, are "pushed" to scientists.

#### Monitoring reproducibility

About 3,600 new publications appear each day on PubMed, in biomedicine alone. Using automated model extension and analysis, ASKE systems can evaluate newly reported mechanisms against experimental observations (data) and vice versa. Reported mechanisms that aren't supported by prior observations, as well as observations that don't make sense with respect to existing models can be detected. This technology can help address some aspects of the reproducibility crisis in a principled way.

#### Automated scientific discovery

There is a large body of unexplained observations (i.e., open scientific questions for which no underlying mechanistic explanation is known) appearing in the biomedical literature and in data stores An ASKE system that immediately aggregates and models new knowledge and evaluates its implications with respect to unexplained observations, is likely to be the first to notice that a previously unexplained observation can now be explained. Novel candidate explanations to observations constructed automatically using ASKE systems can be experimentally confirmed and published.

## 5.4 ASKE Month 9 Milestone Report

### 5.4.1 Generalizing EMMAA: a proof-of-principle model of food insecurity

Until recently, all models in EMMAA represented molecular mechanisms for a given disease or pathway. However, the EMMAA approach can be applied to models in other domains. Conceptually, the EMMAA framework is a good fit for domains where there is a constant flow of novel causal information between interacting "agents" or "concepts" appearing in a structured or unstructured form. To demonstrate the generalizability of EMMAA, we created a model of causal factors influencing food insecurity.

In principle, setting up a new EMMAA model only requires creating a new configuration file that specifies a name, a description, as well as a list of search terms, and any optional arguments used to configure the model building process. In applying EMMAA to a new domain, we extended the set of options that can be specified in the configuration file, including the following:

- The literature catalogue to use to search for new content. Biology models use PubMed (specific to biomedicine), whereas other domain models can now use ScienceDirect (general purpose) to search for new articles.
- The reading system to use to read new text content. The biology models in EMMAA query the INDRA Database each day to search for machine reading extractions for new publications. The Database contains outputs for two biology-specific reading systems (REACH and Sparser) for new daily literature content. Models in other domains can be configured to use the Eidos reading system (via its INDRA interface) to extract a general set of causal relationships between concepts of interest.
- The assembly steps to perform during model extension. We added more granularity to configuration options for the model assembly process, making it possible to apply biology-specific INDRA assembly steps (e.g., protein sequence mapping) only to models where they are relevant.
- The test corpus to use for validating the model. So far, each biology model used the same BEL Large Corpus as a source of test statements to validate against. We made it possible to configure what test corpus to use for a given model, allowing a custom set of relevant tests to be applied to the food insecurity moddel.

To set up the initial, proof-of-principle model of food insecurity, we first identified a set of core concepts of interest: food security, conflict, flooding, food production, human migration, drought, and markets. We then filtered a set of extractions by Eidos on a corpus of 500 documents to causal influences among these concepts. We also set these core concepts as search terms in the model's configuration file. Finally, we defined a set of common sense statements as test conditions, for instance, "droughts cause a decrease in food availability" to check the model against. The model is now included on the EMMAA dashboard where it can be examined (http://emmaa.indra.bio/dashboard/food_insecurity).

While this initial food insecurity model serves as a proof of principle for the generality of the EMMAA concept and the underlying technologies, there are several challenging aspects of building a good model for this domain.

1. The identification of relevant sources of information. So far, the food insecurity model uses ScienceDirect to search for scientific publications. However, it is likely that a significant amount of timely new information is available in reports (by governments, NGOs, etc.) and news stories. In the longer term, this would require implementing ways to query and collect text content from such sources.
2. Querying for relevant text content. We found that certain search terms (e.g., food insecurity) result in mostly relevant publications, while others, wuch as "conflict" or "markets" are too broad and ambiguous, and result in many irrelevant publications being picked up. This suggests that one has to constrain the domain, in addition to the specific concepts used as search terms when finding novel literature content.
3. Machine reading infrastructure. The biology EMMAA models rely on a parallelized AWS infrastructure in which multiple instances of machine reading systems can process hundreds or thousands of new publications each day. In contrast, the food insecurity model currently relies on a single reader instance running as a service, and therefore has much lower throughput. Before a comparable infrastructure of readers is implemented for this domain, we had to limit the number of new publications that are processed each day to update the model.

4. Reading with corroboration. While biology models in EMMAA rely on knowledge assembled from multiple machine reading systems as well as structured (often human curated) knowledge bases, the food insecurity model currently relies on a single reading system, Eidos. This means that any systematic errors specific to the reading system are prone to propagate into the assembled model. In the longer term, integrating more reading systems or knowledge sources could improve on this.
5. Indirect relations. As shown by the initial test set for the food insecurity model, all test statements are satisfied by a single causal influence statement, even ones where one might reasonably expect the test to be satisfied via a chain of causal influences, e.g., "droughts cause a decrease in food availability". We believe that this is due to the fact that authors routinely report indirect causal influences, and the reading/assembly systems currently aren't set up to effectively differentiate between direct and indirect effects.

### 5.4.2 Extending model testing and analysis to multiple resolutions

In our Month 6 Milestone Report, we described an initial experiment to investigate the value of coarse-grained model testing using simple directed graphs. In this reporting period we have extended this concept further by developing a generalized framework for model checking using networks assembled at different levels of granularity and specificity. In particular, we are expanding the range of models assembled from a set of EMMAA Statements to include:

- Directed networks
- Signed directed networks
- PyBEL networks (includes nodes with state information)
- PySB models/Kappa influence maps

For each of these model representations, model checking can be formulated as a process consisting of three steps:

1. Given a (source, target) statement for checking, identify the nodes associated with the source and target. Note that a source or target agent in the test statement may correspond to multiple nodes in the give network representation.
2. Identify causal paths linking one or more source nodes to one or more target nodes. If such a path exists, the test statement is satisfied.
3. Collect paths from the network representation and map them back to the knowledge-level (EMMAA statements) for reporting.

The second step in this process, pathfinding over the causal network, is common to all four of the network representations listed above. However, the first and third steps–identifying mappings between knowledge-level statements and the nodes and edges in the network–are specific to each network representation.

To support multi-resolution model checking we have restructured the INDRA model checker to support multiple model types, with the common code refactored out into a parent class. In addition we have created an assembler that assembles INDRA Statements into a new network representation with a metadata model that can capture the full provenance information from the source INDRA Statements. This network representation, a multi-digraph called the *IndraNet*, will be used to generate multiple coarse-grained "views" (digraph, signed digraph), while preserving statement metadata.

In the upcoming reporting period we will complete this refactoring procedure and extend the EMMAA web application to generate and display test results for alternative realizations of each individual knowledge model.

### 5.4.3 Implementing an object model for model analysis queries

We have previously specified a Model Analysis Query Language (MAQL) used to represent various analysis tasks that can be performed on EMMAA models, in either a user or machine-initiated way (see *Model Analysis Query Language*).

In this reporting period, we implemented a Python object model corresponding to MAQL. The object model provides a structure for all the attributes needed to represent a query, and methods to serialize and deserialize it into JSON. This allows linking the web front-end, the query execution engine, and the back-end query storage database in a principled way through a single standardized format. In particular, we have implemented the PathProperty query class (`emmaa.queries.PathProperty`), and plan to extend to the other three query types specified in MAQL in the coming months.

### 5.4.4 Detecting changes in analysis results due to model updates

One of the fundamental ideas of the EMMAA framework is to be able to detect meaningful changes to analyses of interest as model updates happen. We have implemented an initial solution to this in the QueryManager (`emmaa.answer_queries.QueryManager`) whereby the previous results of each registered query are compared to the new result. Any detected changes are reported in the model update logs (currently not exposed in the user-facing web front-end yet). A limitation of the current approach is that the result of a registered query is a single "top" mechanistic path that satisfies the query conditions, rather than all possible paths. This means that in some cases, when a new path is created by a new piece of knowledge, it would not be detected as a change in the query results, unless the "top" path happens to change. We are planning to improve the change detection method in this direction.

Further, we are working on adding a user registration functionality. Once user accounts and user-specific registered queries are created, the next step will be to create a notification system that exposes the detected changes in analysis results with respect to a query of interest to the user.

## 5.5 ASKE Month 11 Milestone Report

### 5.5.1 Deployment of multiple-resolution model testing and analysis

We have previously described our progress towards developing a capability to check EMMAA models using causal representations at different levels of resolution. During this reporting period we have deployed multiple-resolution model checking for all models hosted in the EMMAA web application. After processing new literature and assembling the corpus of relevant EMMAA statements, the system assembles the knowledge-level information into the following types of causal representations:

- *Unsigned directed networks*. This model type is a simple directed graph with unsigned, directed edges between entities (molecular entities and biological processes in the case of biological networks).
- *Signed directed networks*. Similar to the unsigned, directed network, in that it is a directed graph over entities and processes, but each edge is associated with a sign indicating whether it represents a positive or negative regulation of activity or amount.
- *PyBEL networks*. A PyBEL network is a particular network representation of causal information encoded in the Biological Expression Language (see https://pybel.readthedocs.io). PyBEL networks are also signed and directed, but the nodes in the network have *state*: for a given protein *X*, the mutated, modified, or active forms of *X* are represented by distinct nodes. The inclusion of state information allows the network to represent more specific preconditions for causal influences.
- *PySB models/Kappa influence maps.* In this representation, the EMMAA Statements are used to instantiate a rule-based model using PySB/Kappa, and the Kappa framework is used to analyze the causal structure of the rules in the model. In a Kappa influence map, the nodes are *reaction rules* rather than entities, and each edge reflects the positive or negative influence one reaction rule has on another (for example, if rule A produces P as its product, and P is a precondition for the firing of rule B, the influence map will contain a positive edge between rules A and B). Each rule in the PySB/Kappa model is subject to specific preconditions for activity and hence this representation is the most causally constrained. Until this reporting period, PySB/Kappa models were the only form of model representation subject to automatic testing EMMAA.

Each of these four causal network representations represent entities and causal influences differently; the first step in automated checking of causal queries is therefore to ground the entities in the *query* to nodes in the particular network representation. For example, in the causal query "How does phosphorylated BRAF increase MAPK1 activity", the subject node is "phosphorylated BRAF" and the object node is "MAPK1 activity" (Figure 1). In the unsigned and signed directed networks, these two concepts map simply to the nodes for BRAF and MAPK1, because these networks do not distinguish based on entity state. In the PyBEL network, there are multiple nodes consistent with "phosphorylated BRAF", including *p(BRAF, pmod(P, S, 602))* (representing BRAF phosphorylated at serine 602) and *p(BRAF, pmod(P))* representing BRAF phosphorylated at an unknown site; similarly, there are multiple nodes corresponding to "MAPK1 activity", including *act(MAPK1)* and *kin(MAPK1)*, representing the generic molecular and specific kinase activity of MAPK1, respectively. For the PySB/Kappa influence map, there are multiple rules consistent with phosphorylated BRAF as source nodes, and multiple observables corresponding to MAPK1 being in a state consistent with its activity. Checking the model involves identifying these subject and object nodes and then searching for paths linking any subject node to any object node. If any such path is found, then this represents a candidate causal explanation in that representation.

"How does phosphorylated BRAF increase MAPK1 activity?"

| | "phosphorylated BRAF" | "active MAPK1" |
|---|---|---|
| Unsigned Graph | BRAF | MAPK1 |
| Signed Graph | BRAF | MAPK1 |
| BEL Graph | p(BRAF, pmod(P, S, 602))<br>p(BRAF, pmod(P))<br>p(BRAF, pmod(P, T, 599)) | act(MAPK1)<br>kin(MAPK1) |
| Influence Map | Rules with phosphorylated BRAF as *subject* on LHS | Observables with MAPK1 in phosphorylated state |

*Figure 1: Network nodes associated with the subject and object of the causal query "How does phosphorylated BRAF increase MAPK1 activity?" using the four causal representations deployed in this reporting period.*

In addition to generating the model testing results on the back end, the EMMAA web application now presents the results of multi-resolution model checking to the user. The *Tests* tab of the model landing page now highlights the proportion of passed tests for each model type (Figure 2). As expected, the least stringent causal representation (unsigned graph) generally yields the highest proportion of passing tests, while the most stringent (PySB) yields the lowest.

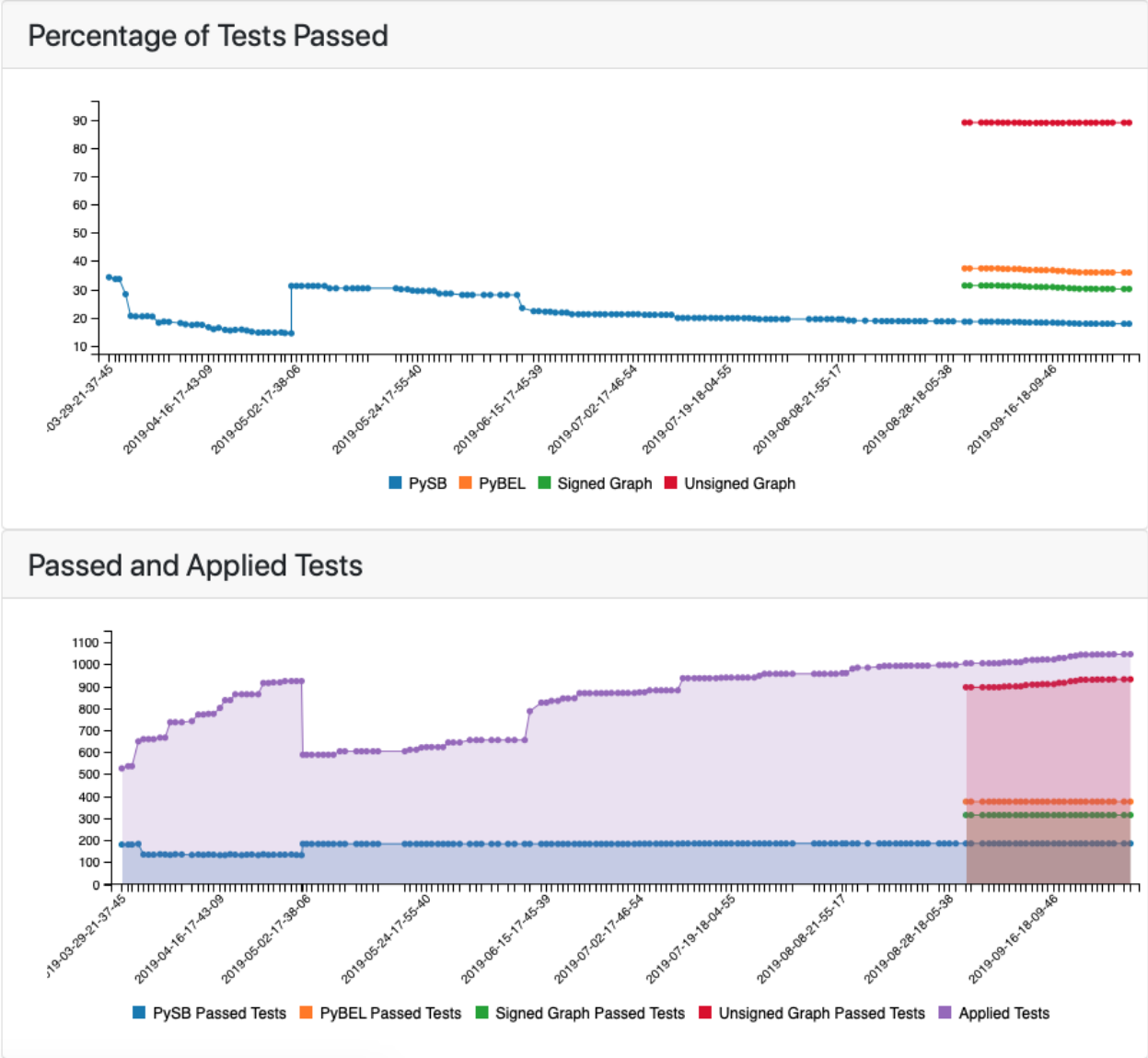



*Figure 2: Test report graph highlighting the percentage of applied tests passed in each of the four causal representations.*

In addition, the test report page now displays tests results as a matrix rather than a simple list (Figure 3). Each icon is hyperlinked to a test details page showing information about the test and the causal paths found to explain the causal query.

All Test Results

| Test | Pysb | Pybel | Signed Graph | Unsigned Graph |
|---|---|---|---|---|
| AGT activates EGFR. | ✗ | ✓ | ✓ | ✓ |
| AGT activates RAS. | ✗ | ✓ | ✓ | ✓ |
| AGT bound to AGTR1 activates ERK. | ✗ | ✗ | ✓ | ✓ |
| Kinase-active AKT activates ESR2. | ✗ | ✗ | ✗ | ✓ |
| Kinase-active AKT activates angiogenesis. | ✗ | ✗ | ✓ | ✓ |
| Kinase-active AKT activates cell differentiation. | ✗ | ✗ | ✓ | ✓ |
| AKT activates MTOR. | ✗ | ✓ | ✓ | ✓ |
| ANGPT1 activates RAS. | ✗ | ✓ | ✓ | ✓ |
| ANGPT1 activates angiogenesis. | ✗ | ✗ | ✓ | ✓ |
| Kinase-active BRAF activates ERK. | ✓ | ✓ | ✓ | ✓ |
| GTP-bound active CDC42 activates MTOR. | ✗ | ✗ | ✗ | ✓ |
| CDKN2A activates TP53. | ✓ | ✓ | ✓ | ✓ |

*Figure 3: Test result matrix with the green and red icons indicating whether the given test passed or failed in the specific model representation, respectively.*

### 5.5.2 User-specific query registration and subscription

We implemented a user registration and login feature in the EMMAA dashboard which allows registering and subscribing to user-specific queries. After registering an account and logging in, users can now subscribe to a query of their interest on the EMMAA Dashboard's Queries page (https://emmaa.indra.bio/query). Queries submitted by users are stored in EMMAA's database, and are executed daily with the latest version of the corresponding models. The results of the new analysis are then displayed for the user who subscribed to the query on the query page. This allows users to come back to the EMMAA website daily, and observe how updates to models result in new analysis results. Later, we are planning to report any relevant change to the analysis results directly to the user by sending a notification via email or Slack.

This capability is one important step towards achieving "push science" in which users are notified about relevant new discoveries if the inclusion of these discoveries result in meaningful changes in the context of prior knowledge (i.e., a model) with respect to a scientific question.

### 5.5.3 An improved food insecurity model

This month we migrated the food insecurity model to use the new World Modelers ontology (https://github.com/WorldModelers/Ontologies), and expanded its set of search terms. This significantly increased the models' size and the granularity of concepts over which it represents causal influences:

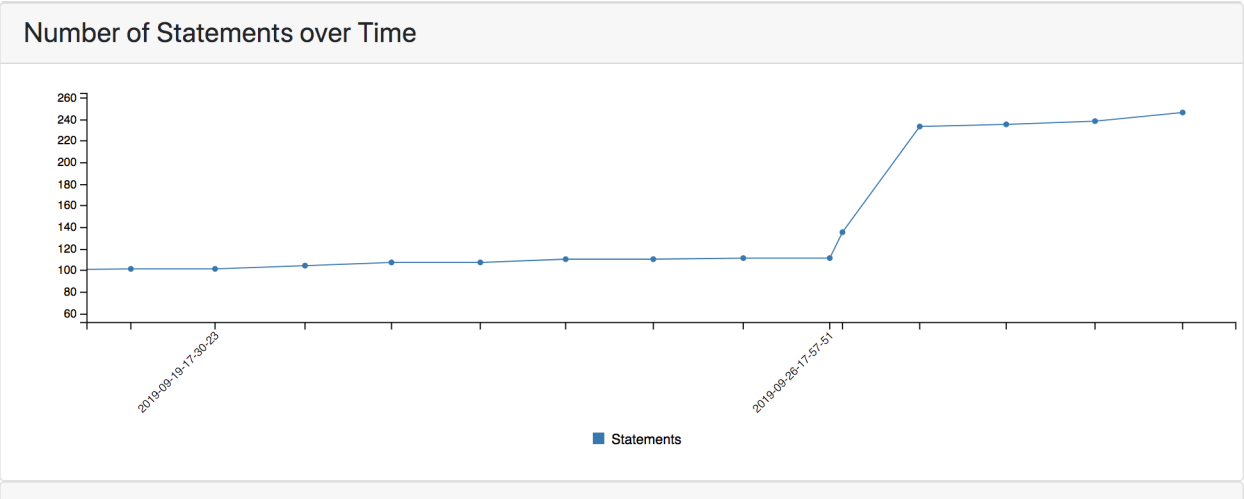



*Figure 4: Size of the food insecurity model over time.*

Below is a snapshot of the network view of the model on NDEx (https://ndexbio.org/#/network/478a3ed6-b3b7-11e9-8bb4-0ac135e8bacf):

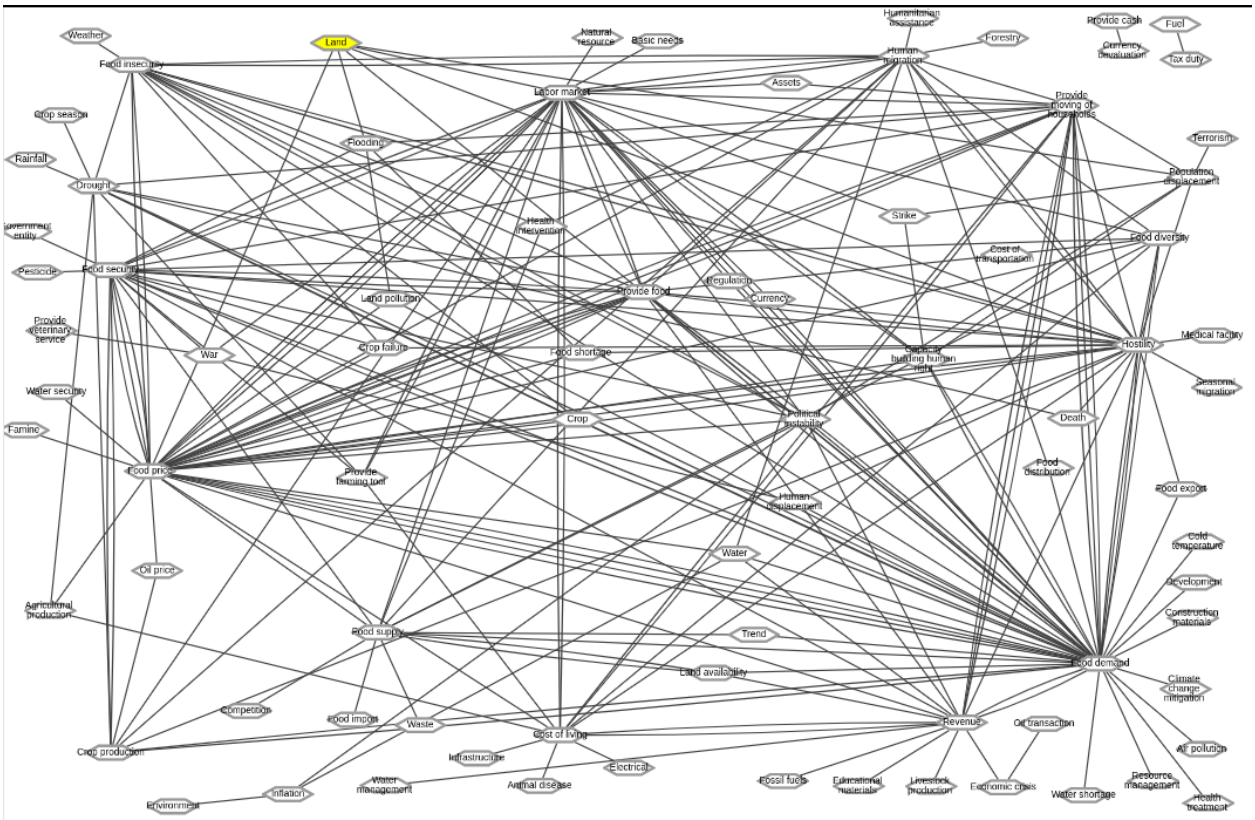



*Figure 5: Snapshot of the food insecurity model as of 9/27/2019.*

## 5.6 ASKE Month 13 Milestone Report

### 5.6.1 Related work for the EMMAA system

We are not aware of any meta-modeling systems coupling machine-assembled models to automated analysis, in molecular biology or other fields. To the best of our knowledge, the EMMAA system is the first of its kind. Despite the fact that EMMAA is unique as an integrated system, there does exist a body of pre-existing work related to individual component technologies of the system.

Mathematical and causal modeling has been widely applied in systems biology, where a multitude of model types (ordinary and partial differential equations, Boolean and logical models, probabilistic graphical models, etc.) have been used to represent the behavior of biochemical mechanisms (Aldridge et al., 2006). However, such models are difficult and time consuming to build, and require special mathematical and computational expertise. To address this, EMMAA draws on novel tools allowing the automated assembly of mathematical models directly from text (INDRA; Gyori et al., 2017).

There also exists a large body of work in text mining in biomedicine (Ananiadou et al., 2006), motivated by the fact that around 3,200 new publications appear every day - too much for any human expert to keep up with. However, the output of these systems have thus far not been combined (EMMAA currently integrates and aligns output from 4 different text mining systems: REACH (Valenzuela-Escárcega et al., 2019), Sparser (McDonald et al., 2016), TRIPS/DRUM (Allen et. al., 2015) and RLIMS-P (Torii et al., 2015)) and made available for natural language querying by users. Recently, a graphical user interface was proposed to explore causal relations extracted by a single reading system (Barbosa et al., 2019). However, the causal networks built using this system do not make use of the knowledge assembly procedures built into EMMAA, including correction of systematic reading errors, and assessment of redundance, relevance, and believability.

Further, several large human-curated knowledge-bases for molecular mechanisms have been developed (Cerami et al, 2010, Croft et al., 2013), and can be queried through their respective websites through standard web forms. Finally, large repositories of experimental and clinical data are routinely used in biomedicine (Keenan et al., 2018, Tomczak et al., 2015). However, while such repositories exist, they grow only through manual curation and are often out of date.

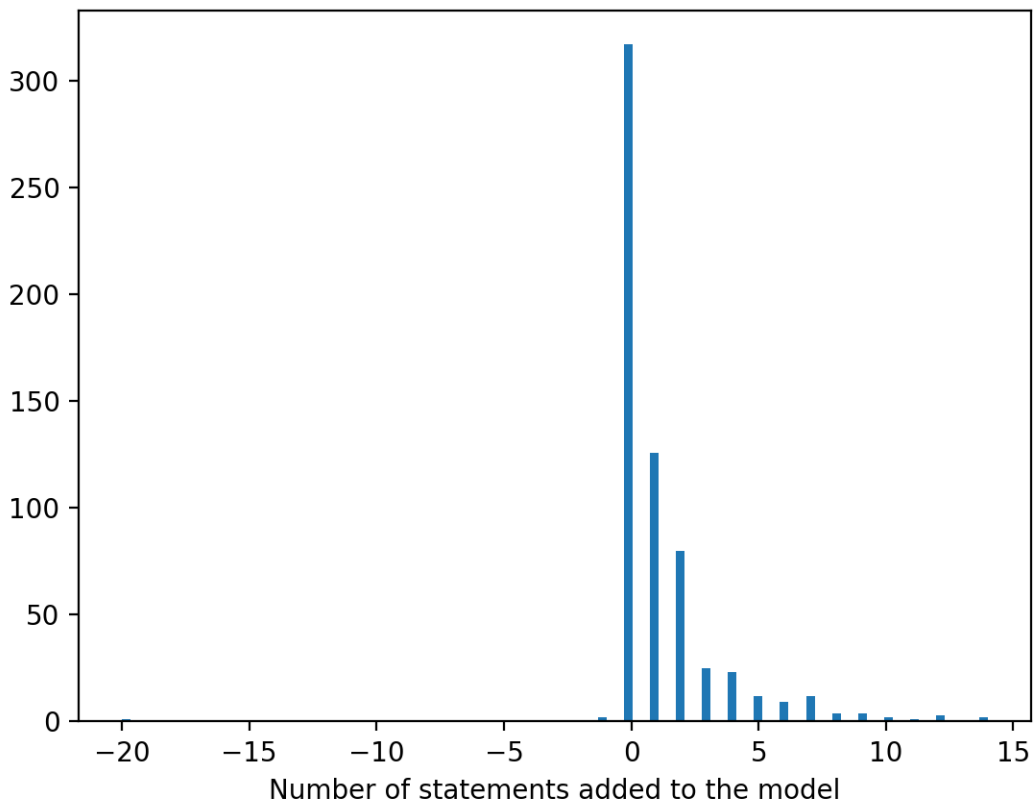
Finally, while the concept of model testing and validation, either static or dynamic, is not new, this has (to our knowledge) only been applied to specific models in isolated modeling studies. There exists no framework for the systematic evaluation of domain models with respect to relevant tests; nor are there any previous demonstrations of the use of text mining to automatically grow a body of observations for use in model evaluation.

### 5.6.2 System performance statistics

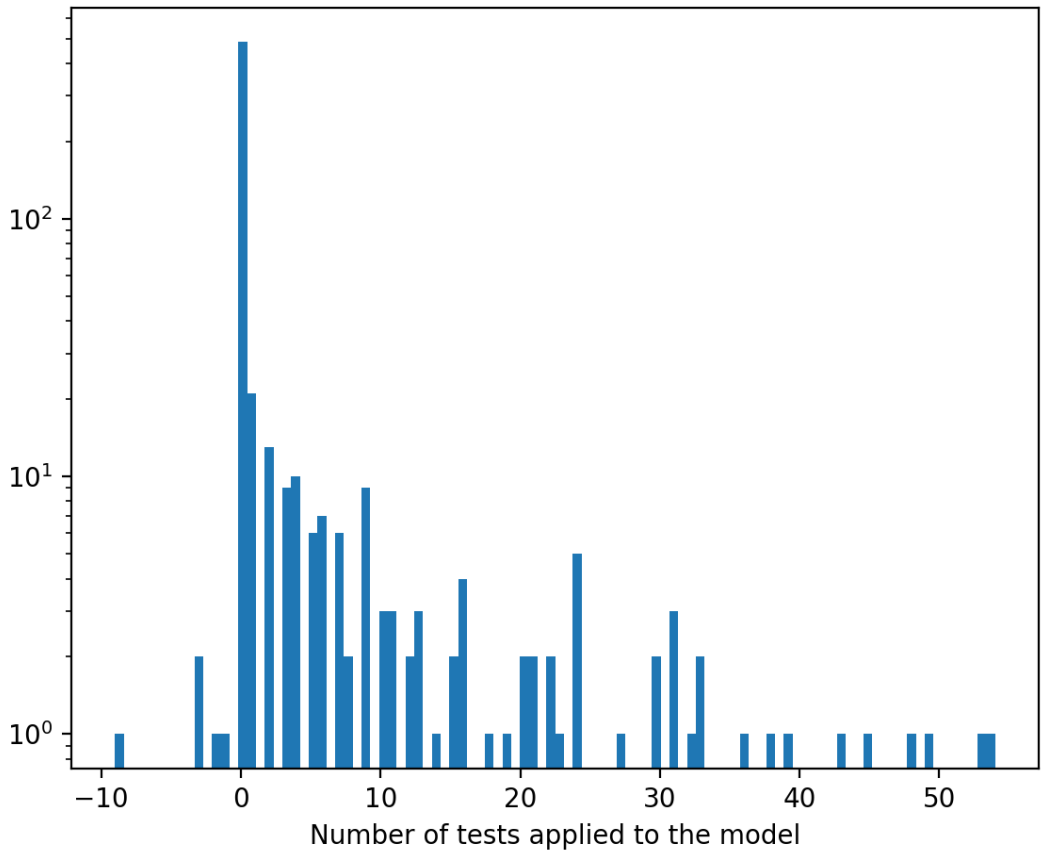
EMMAA currenty manages a total of 11 models. Eight of these models are fully machine-maintained and represent various diseases (7 models) and pathways (1 model). Two models are based on expert-curated natural language, then linked to literature evidence and tested automatically. Finally, one model represents a set of causal factors affecting food insecurity, i.e., is outside the domain of molecular biology.

To quantify the performance of the system in terms of extending and testing/ analyzing models, we plotted the distribution of (1) number of new statements added (2) number of new tests applied and (3) change in the test pass ratio for each of the machine-maintained biology models each day.

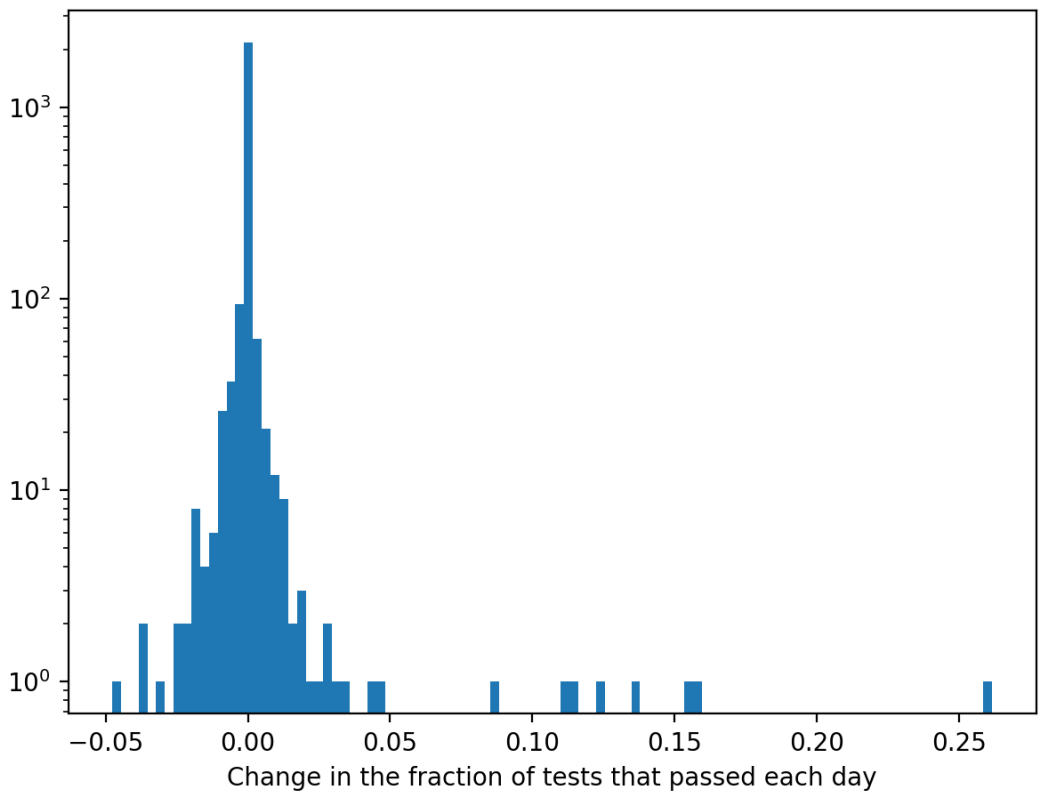
Histogram of the number of new statements added to each model each day. As we can see, the change in the number of statements is often zero (i.e., no new mechanisms were found relevant to the given model), but otherwise is between 1-15 new statements per day. In some cases, the assembly procedure removes previously existing mechanisms from the model, thereby making the number of statements added negative.




Histogram of the number of new applied tests each day. Typically, if new statements are added to a model, the number of applied tests can increase. As shown in the histogram, new mechanisms added to a model often result in dozens of new test being applicable to the model.




Histogram of the change in the fraction of tests that pass (across all four modeling formalisms, PySB, PyBEL, signed graph, unsigned graph) each day compared to the previous day. While small fractional changes are more common, in some cases, model extensions (or changes to model assembly) resulted in large jumps in test pass ratio of 5-25%.




### References

Aldridge, B. B., Burke, J. M., Lauffenburger, D. A., & Sorger, P. K. (2006). Physicochemical modelling of cell signalling pathways. Nature cell biology, 8(11), 1195.

Gyori, B. M., Bachman, J. A., Subramanian, K., Muhlich, J. L., Galescu, L., & Sorger, P. K. (2017). From word models to executable models of signaling networks using automated assembly. Molecular systems biology, 13(11).

Ananiadou, S., & McNaught, J. (2005). Text mining for biology and biomedicine (pp. 1-12). London: Artech House.

Valenzuela-Escárcega, M. A., Babur, Ö., Hahn-Powell, G., Bell, D., Hicks, T., Noriega-Atala, E., … & Morrison, C. T. (2018). Large-scale automated machine reading discovers new cancer-driving mechanisms. Database, 2018.

McDonald, D., Friedman, S., Paullada, A., Bobrow, R., & Burstein, M. (2016, March). Extending biology models with deep NLP over scientific articles. In Workshops at the Thirtieth AAAI Conference on Artificial Intelligence.

Allen, J., de Beaumont, W., Galescu, L., & Teng, C. M. (2015, July). Complex Event Extraction using DRUM. In Proceedings of BioNLP 15 (pp. 1-11).

Torii, M., Arighi, C. N., Li, G., Wang, Q., Wu, C. H., & Vijay-Shanker, K. (2015). RLIMS-P 2.0: a generalizable rule-based information extraction system for literature mining of protein phosphorylation information. IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB), 12(1), 17-29.

Barbosa, G. C., Wong, Z., Hahn-Powell, G., Bell, D., Sharp, R., Valenzuela-Escárcega, M. A., & Surdeanu, M. (2019, June). Enabling Search and Collaborative Assembly of Causal Interactions Extracted from Multilingual and Multi-domain Free Text. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations) (pp. 12-17).

Cerami, E. G., Gross, B. E., Demir, E., Rodchenkov, I., Babur, Ö., Anwar, N., … & Sander, C. (2010). Pathway Commons, a web resource for biological pathway data. Nucleic acids research, 39, D685-D690.

Croft, D., Mundo, A. F., Haw, R., Milacic, M., Weiser, J., Wu, G., … & Jassal, B. (2013). The Reactome pathway knowledgebase. Nucleic acids research, 42(D1), D472-D477.

Keenan, A. B., Jenkins, S. L., Jagodnik, K. M., Koplev, S., He, E., Torre, D., … & Kuleshov, M. V. (2018). The library of integrated network-based cellular signatures NIH program: system-level cataloging of human cells response to perturbations. Cell systems, 6(1), 13-24.

Tomczak, K., Czerwińska, P., & Wiznerowicz, M. (2015). The Cancer Genome Atlas (TCGA): an immeasurable source of knowledge. Contemporary oncology, 19(1A), A68.

## 5.7 ASKE Month 15 Milestone Report

### 5.7.1 EMMAA Knowledge assemblies as alternative test corpora

During this reporting period we have made two significant updates to our approach to static analysis of models against observations. First, we have implemented a prototype capability to generalize EMMMAA knowledge assemblies for use as either models or as tests. Second, we have implemented the capability to test a single model against multiple corpora, which involved changes to both the back-end test execution as well as the user interface for displaying test results.

In EMMAA, daily machine reading is used to update a set of causal relations relevant to a specific domain, such as a disease, signaling pathway, or phenomenon (e.g., food insecurity). Up until this point, these (possibly noisy) knowledge assemblies have been used to build causal models that are checked against a set of manually-curated observations. We have now also implemented the converse procedure, whereby the knowledge assemblies are treated as sets of *observations*, used to check manually curated models.

A prerequisite for this approach is the ability to run a single model against alternative test suites, which required significant refactoring of our back-end procedures for triggering testing and results generation, and new user interfaces to display multiple test results. This feature is described in the documentation for the *Tests Tab*.

As a proof of concept, we converted the EMMAA Statements used to generate the Ras Machine 2.0 (*rasmachine*) and Melanoma (*skcm*) models into sets of EMMAA Tests, and checked the manually-curated Ras Model (*rasmodel*) against each set independently. The user can now choose between these alternative test corpora in the EMMAA user interface:

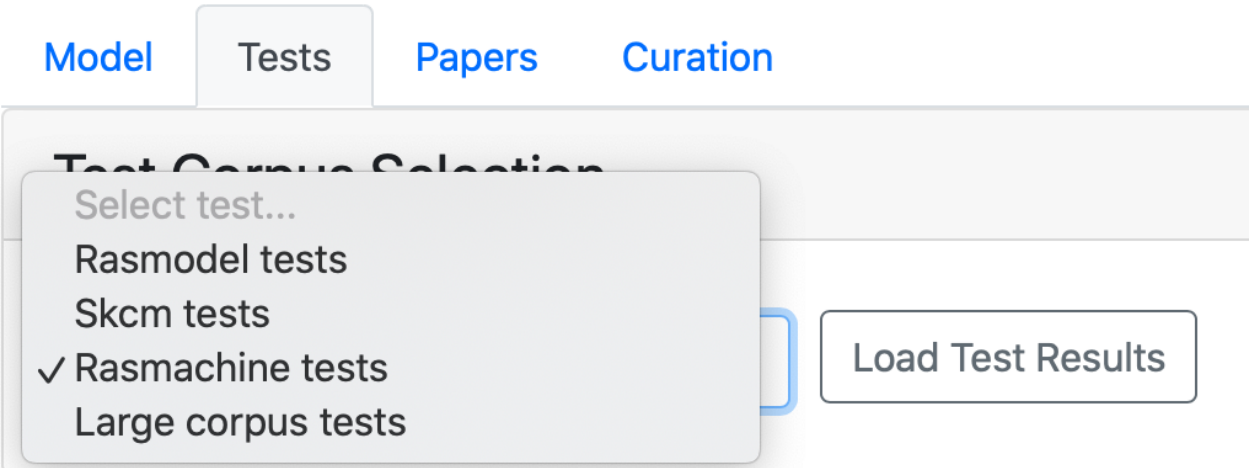



Fig. 1: *Selecting test results to view among "Large Corpus Tests", "Rasmachine Tests" and "Skcm tests".*

Examining the performance of the curated Ras Model against these three different corpora reveals striking differences. The PySB implementation of the Ras Model has a passing rate of 55% for the BEL Large Corpus (100/182 tests), but only 16% (120/730 tests) for the Ras Machine test corpus and 7% (6/86 tests) for the Melanoma test corpus. We inspected a handful of the tests from the Ras Machine that the Ras Model did not pass. Many of these failed tests highlighted aspects of the Ras Model that were failing either for minor technical reasons (e.g., "CCND1 activates CDK4", which failed due to the active form of CDK4 being defined explicitly in the model); others represented knowledge gaps that could guide additions to the model (e.g., "RPS6KA1 activates RPTOR"). This latter category represent an opportunity for *test-driven modeling* as we described in an earlier report, with the additional feature that here the system is *automatically* providing guidance for model extension based on ongoing mining of the literature.

In addition, we also found a number of cases where the failure of the Ras Model to pass a test highlighted errors in the underlying machine reading underlying the test. For example, the Melanoma Model included the test "PTEN ubiquitinates PTEN", which was derived from jointly incorrect extractions from three distinct sentences. As the Ras Model is extended to cover more of the true biology of the Ras pathway, we anticipate that failed tests will be increasingly likely to be erroneous. From a larger perspective, we believe that this approach highlights the prospect of using causal models to determine the *a priori* plausibility of a newly-reported finding extracted by text mining.

### 5.7.2 Time machine

When EMMAA performs daily updates, it reports which new statements were newly added to each model, the new tests that were applied based on the these statements, and whether these new tests passed or failed. Until this point the user could only see the change in statements and tests from the most recent update. This prevented the user from investigating the changes at earlier points in time, for example at points where there were large changes in the number of tests passing. During this reporting period we have added a "time machine" feature to the user interface to allow the user to inspect changes in the model statements and tests at specific previous timepoints.

For example, the history of the Ras Machine model shows that on 11/26/2019, there was a dramatic change in the pass ratio of PyBEL model tests, as shown below:

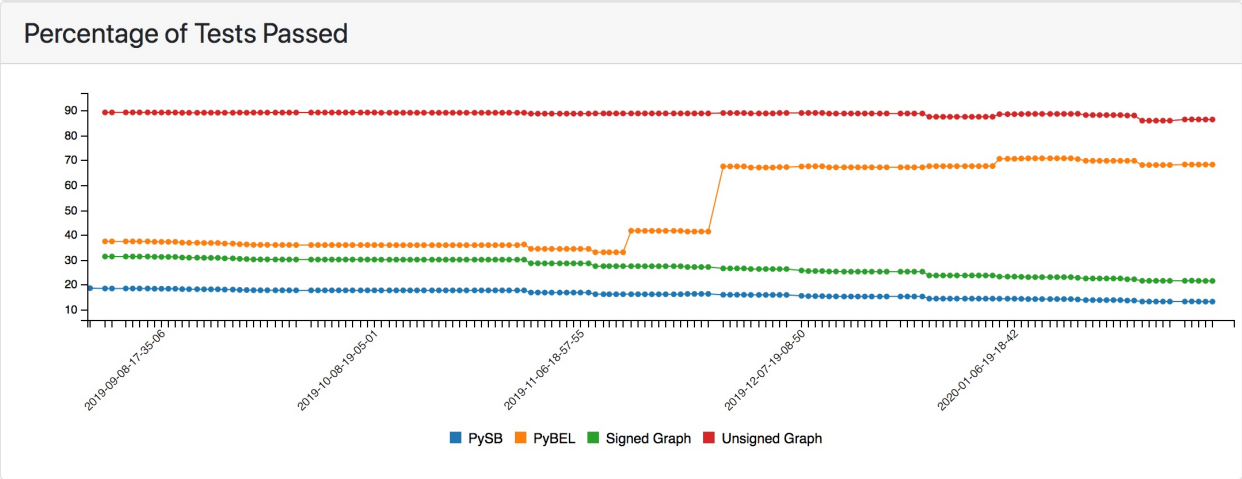



Fig. 2: *Substantial change in the PyBEL pass ratio for the Ras Machine model on November 26, 2019.*

Clicking on the timepoint after the change refreshes the interface to display which tests were newly passed at this point:

Inspection of these newly passed tests along with the changes in model statements can help the user understand changes in the causal structure of the model over time.

This feature is described in the documentation section *Load Previous State of Model*.

New Passed Tests

| Test | Top Path |
|---|---|
| **New passed tests for PyBEL model.** | |
| AKT inhibits CDKN1B. | Path found but exceeds search depth |
| Catalytically active PTGS2 increases the amount of CCND1. | Path found but exceeds search depth |
| MIR21 increases the amount of MMP2. | MIR21 → MIR21 → PTEN → MMP2 → MMP2 |

### 5.7.3 Dynamical model simulation and testing

Initially, the EMMAA project focused on a single mode of model analysis: finding mechanistic paths between a source (e.g., a perturbed protein) and a readout. This mode of analysis is static in that it relies on the causal connectivity structure of the model to characterize its behavior.

We have generalized EMMAA model analysis to dynamical properties in which model simulation is performed. First, EMMAA Statements are assembled into a PySB model - a rule-based representation from which a reaction network, and subsequently, a set of coupled ordinary differential equations (ODEs) can be generated. Given suitable parameters and initial conditions, this set of ODEs can be solved numerically to reconstruct the temporal profile of observables of interest.

Our goal was to design a simple specification language that allows a user to choose an observable, and determine whether it follows a given dynamical profile of interest. An example could be: "In the RAS model, is phosphorylated ERK transient?". Here "phosphorylated ERK" is the observable, and "transient" is the dynamical profile. The user can choose from the following dynamical profiles:

- always value (low/high): the observable is always at a given level
- sometime value (low/high): at some point in time, the observable reaches the given level
- eventual value (low/high): the observable eventually reaches a given level and then stays there
- no change: the observable's level never changes
- transient: the observable's level first goes up and then goes back down
- sustained: the observable's level goes up and then stays at a high level

Internally, EMMAA uses a bounded linear-time temporal logic (BLTL) model checking framework to evaluate these properties. BLTL is defined over discrete time and so we choose a suitable sampling rate at which the observable's time course profile is reconstructed. A temporal logic formula is then constructed around atomic propositions to represent the query. Each atomic proposition has the form [observable,level] and evaluates to True if the observable is at the given level at the current time point. Atomic propositions are then embedded in formulae using standard BLTL operators including X, F, G and U, combined with standard logical operators (~, ^, v). For instance, "is phosphorylated ERK transient?" would be turned into the BLTL property [pERK,low]^F([pERK,high])^F(G([pERK,low])), which can informally be interpreted as: "pERK is initially low, after which at some point it reaches a high level, after which is goes to a low level and remains there."

Given a model simulation, a generic BLTL model checker takes the simulation output (for the observable) and determines whether it satisfies the given formula. The result (pass/fail) is then displayed on the dashboard along with a plot of the actual simulation.

In the future, we plan to account for the parameteric (and potentially the structural) uncertainty of each model using sampling, and use statistical model checking techniques with given false positive and false negative guarantees to produce a pass/fail result.

This feature is described in *Dynamical Queries*.

### 5.7.4 Towards push science: User notifications of newly-discovered query results

The system of user notifications is an important component of the EMMAA concept. As a first approach, we implemented a registration system for users so that when a registered user logs in, they can register specific queries that they are interested in monitoring over time.

Currently, the Query page allows users to browse the results of their registered queries given the current state of each model for which the query is registered. Independently, EMMAA's *answer_queries* module can detect if the result of a registered query changes due to a model update. Putting these two capabilities together, we developed a user notification system in EMMAA. If a specific model update changes the result of a registered user query, the user receives an email notifying them about the change. Importantly, the change to model behavior is attributable to the most recent model update (in which a new discovery from literature was assembled into the model). This creates a system in which new research results, as soon as they are published, are integrated into models that are then evaluated with respect to specific analyses, and their effect on model behavior is assessed and exposed to users whose research it affects. The email notification system is currently being tested internally, and will be exposed on the public interface in the next reporting period.

## 5.8 ASKE Month 18 Milestone Report

### 5.8.1 Expert curation of models on the EMMAA dashboard

Previously, statements constituting each EMMAA model were linked to an outside website (the INDRA DB) where they could be curated by users as correct or incorrect. However, this feature was not convenient for at least two reasons: the curation required moving to an external website, and the specific scope and state of each individual EMMAA model was not always correctly reflected on the more generic INDRA DB site.

Therefore, we implemented several new features in EMMAA that allow curating model statements (and model tests) directly on the dashboard. Some of the key places that allow curation include

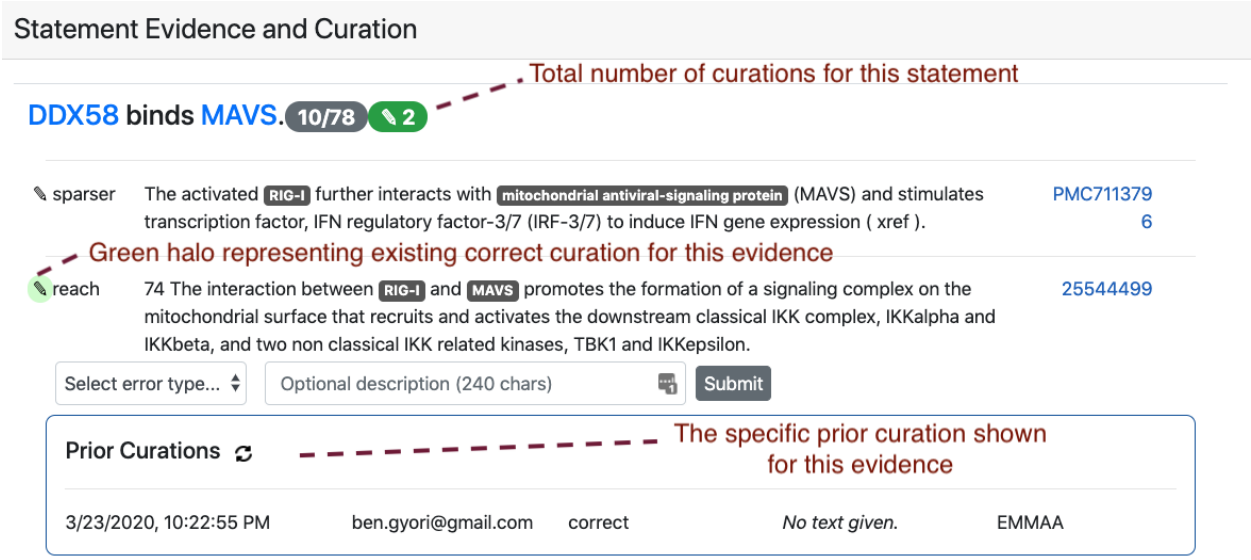
- The list of most supported statements on the Model tab.
- The list of new added statements on the Model tab.
- The page where all statements in a model can be browsed.
- Each model's Test tab allows curating tests themselves (which in some cases are also prone to errors) and also the results of test, i.e., paths of mechanisms satisfying the test.
- Results of new queries and registered queries on the Queries page.

Existing curations for all of the above content are also accessible within the dashboard.

The figure below shows an example of the interface for entering new curations. as well as the visual annotations used to show existing curations and their properties for each statement or evidence.

### 5.8.2 Viewing and ranking all statements in a model

We also recognized the importance of being able to inspect the contents of the model as a whole, in a view which exposes all the literature evidence and also enables in-place curation (as opposed to the NDEx network view). Therefore, we added a "View All Statements" button to each Model page which allows browsing all statements in the model. To overcome the challenge of the model potentially containing a very large number of statements, the page uses an auto-expand feature which loads statements in real time as the user scrolls further down on the page. Similarly, evidences for each statement are loaded during runtime, and only when requested by the user.




The default view on the All Statements page ranks statements by the number of evidence that support them. This allows curators to focus on statements that are prominently discussed in the subset of literature corresponding to the scope of the model. However, this ranking doesn't necessarily correspond to a statement's importance in terms of functionally affecting a model's behavior. Therefore, we added another option to sort statements by the number of model tests whose result rely on the statement. In other words, if a given mechanism is essential for many tests passing, it will be ranked high on this page. This view is particularly useful if a user intends to curate the model in a way that they focus on identifying incorrect statements that have the biggest functional effect on model behavior, without spending time on statements that do not play an important role in this sense.

### 5.8.3 Email notifications

The system of user notifications for registered queries is now in place and available to any registered user. On the Query page, when a query is registered, the user is also signed up for email notifications. This means that each time a relevant new result is available for the query, the user receives an email informing them what the new result is, and linking them to the page on which the new result and its effect on model behavior can be inspected.

A representative use case for this is a query about a drug and an indirect downstream effect that could be explained by many possible parallel paths of mechanisms (e.g., "how does quercetin inhibit TMPRSS2?"). Each day, as a model is updated, new mechanisms that were extracted from the latest literature may provide links between previously unconnected concepts that can contribute to new results for a query.

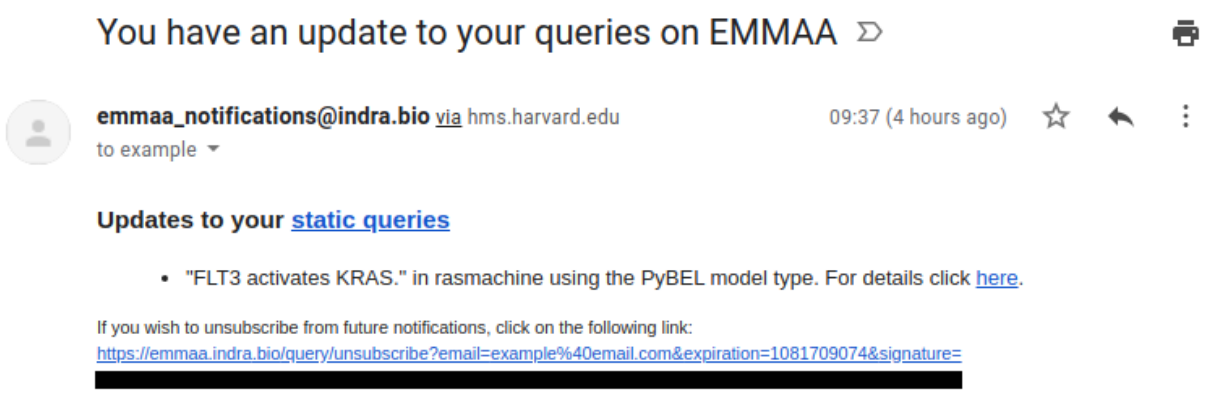
The figure below shows an example notification email that an EMMAA user would receive:

### 5.8.4 A model of Covid-19

Before starting the project, we had planned to set up at least one EMMAA model of a relevant public health-related process. As the Covid-19 crisis emerged, we set up an EMMAA model (https://emmaa.indra.bio/dashboard/covid19/?tab=model) to capture the relevant existing literature (by building on the CORD-19 corpus). The model also self-updates each day with new literature on Covid-19, which is now appearing at a pace of ~500 papers a day, and accelerating.

We have made a number of enhancements to the underlying reading and assembly pipelines to:

1. Incorporate full text content from the CORD-19 corpus alongside our other sources (PubMed Central, MEDLINE, Elsevier, xDD)




2. Improve grounding of viral proteins, e.g., "SARS-CoV-2 Spike protein"
3. Use GILDA (https://github.com/indralab/gilda) to ground named entities identified by the University of Arizona open-domain reading system Eidos to extract and integrate high-level causal relations (e.g., viruses cause respiratory inf.

In addition, we have added curated tests describing empirically observed inhibitors of SARS-CoV-2 (e.g., "Imatinib methanesulfonate inhibits severe acute respiratory syndrome coronavirus 2") to determine whether the model can identify mechanistic explanations for the effectiveness of these drugs.

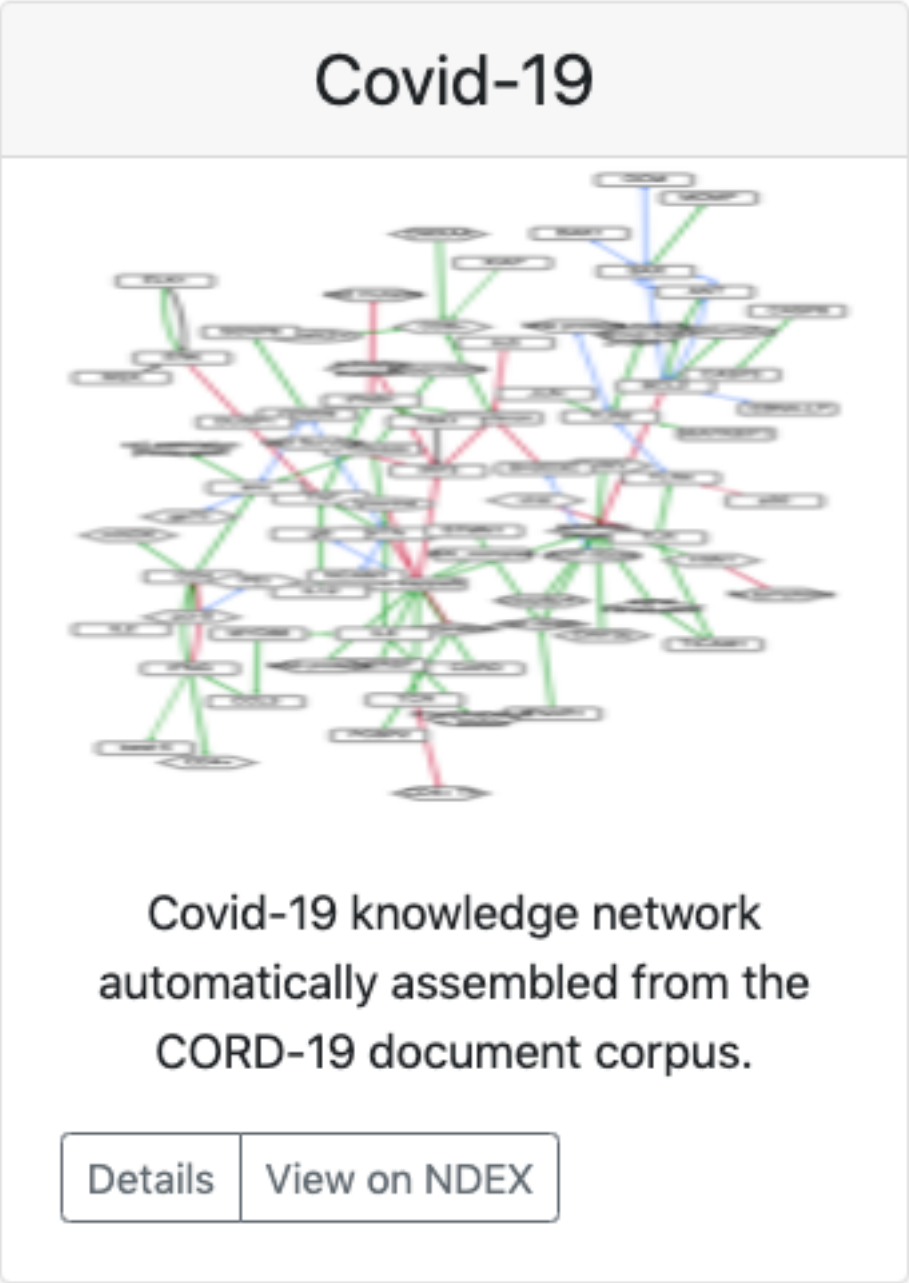
### 5.8.5 Integration of content from UW xDD system

During this reporting period we have continued to develop our pipeline to integrate content from the University of Wisconsin xDD platform. To support the pipeline we have created new command-line endpoints to run machine reading and statement extraction within our Dockerized system. INDRA Statements extracted from the xDD content are posted to a shared private AWS S3 bucket along with associated document metadata. In five successive pilot runs we have refined metadata formats and adapted the schema of the INDRA DB to allow INDRA Statements to be linked to article metadata in the absence of article content (we only obtain INDRA Statements from xDD, while xDD retains the articles themselves). Next steps include:

1) Determining relevance of xDD documents to specific EMMAA models by linking documents to specific xDD-indexed terms/keywords
2) Scaling up to larger document runs focusing on Pubmed-indexed documents for which we do not have full texts available from other sources.

### 5.8.6 Configurable model assembly pipeline

Each EMMAA model is defined by a configuration file which determines what search terms the model is built around, other metadata (name, description etc.), and other settings specific to the model. Building on the new Pipeline feature in INDRA, EMMAA models can now define the assembly pipeline applied to each model in a fully declarative way, as part of the configuration file. This simplifies the EMMAA codebase, and makes the instantiation of new models much easier, in a way that is decoupled from code. This could open up exciting possibilities such as instantiating EMMAA models on-demand, potentially through a UI.


Covid-19
Covid-19 knowledge network automatically assembled from the CORD-19 document corpus.
Details
View on NDEX

# CHAPTER 6

# ASKE-E Reports

This section contains reports on the EMMAA project as part of the DARPA Automating Scientific Knowledge Extraction (ASKE) program extension.

## 6.1 ASKE-E Month 1 Milestone Report

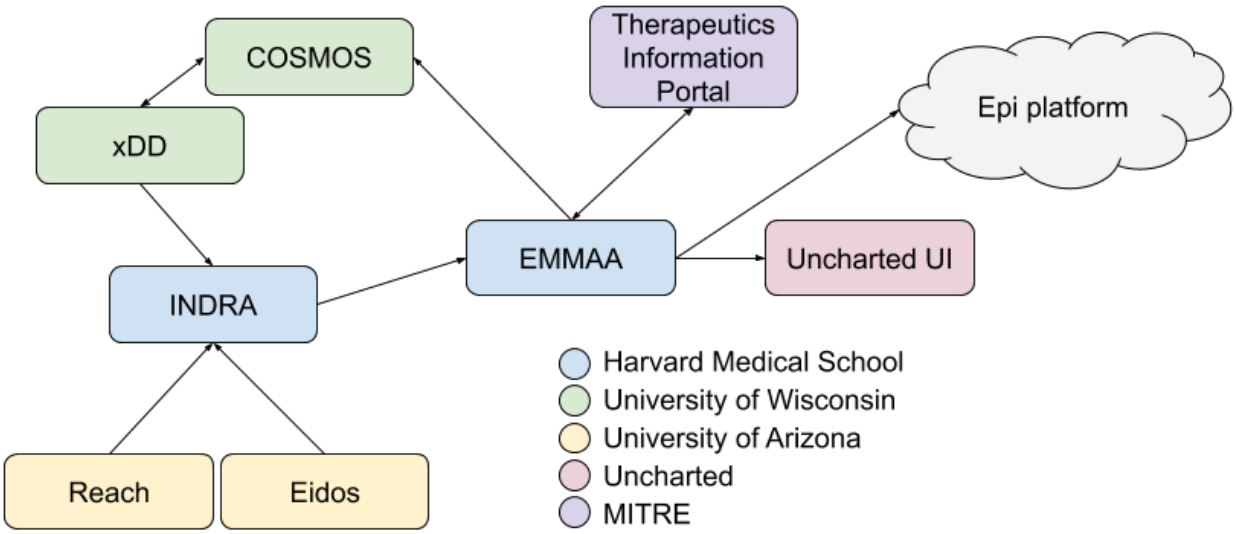
### 6.1.1 Overall goals and use cases for the Bio Platform

The goal of the Bio Platform is to provide an automated modeling and model analysis platform (with appropriate interfaces for user-in-the-loop interaction) around the molecular basis of diseases and their therapies. The initial disease focus for the platform is COVID-19. In this context, the use cases we aim to work towards are as follows:

- Explain drug mechanisms based on existing experimental observations
  - Example: through what mechanism does E64-D decrease SARS-CoV-2 replication?
- Propose new drugs that haven't yet been tested
  - Example: Leupeptin should be investigated since through protease inhibition, it is expected to decrease SARS-CoV-2 entry.
- Causally/mechanistically explain high-level/clinical associations that are unexplained
  - Example: What is the mechanistic basis for men being susceptible to more severe COVID-19 compared to women?
- Construct reports on the implication of therapeutics on clinical outcomes, optimize course of therapy
  - Example: Find the optimal course of interferon treatment using modeling and simulation.

### 6.1.2 Integration plan for the Bio Platform

The following diagram shows the integration architecture for the Bio Platform:




The main components of this integration are as follows. The HMS team's INDRA system integrates multiple knowledge sources, including the Reach and Eidos machine-reading systems developed by the UA team. INDRA is also integrated with UW's xDD system where it is run on a subset of published papers and preprints to produce statements that INDRA doesn't otherwise have access to. xDD will also provide provenance information for relevant figures and tables coupled to statement evidences.

INDRA produces statements daily that are picked up by EMMAA (each EMMAA model gets only statements that are specifically relevant to its use case as controlled by a definition of model scope). Each EMMAA model then assembles its statements in a use-case-specific way to produce an assembled knowledge base. This is then the basis of generating multiple executable / analyzable model types (unsigned graph, signed graph, PyBEL, PySB) and applying these models to automatically explain a set of observations (note that this process can also be thought of as "testing" or "validation" of the model).

EMMAA integrates with the MITRE Therapeutics Information Portal by pulling in observations about drug-virus relationships that it then explains. The resulting explanations (typically mechanistic paths) will be linked back to the MITRE portal. The portal will also link to INDRA-assembled information on specific drugs and their targets.

EMMAA models will also link back to UW's COSMOS system to provide additional annotations for documents they index.

EMMAA will integrate with the Uncharted UI both at the level of the knowledge base that each model constitutes, and the explanations produced by each model.

Finally, the COVID-19 EMMAA model will also attempt to form links with the Epi Platform by using causal relations between molecular and high-level (e.g., clinical, epidemiological) factors to connect therapeutic interventions to epidemiology.

### 6.1.3 Progress during the ASKE-E Hackathon

Our teams made progress on multiple fronts during the first ASKE-E Hackathon.

First, with the UA team, we identified relevant resources for the lexicalization of protein fragments. The initial goal was to identify and extract relevant terms from the Protein Ontology (https://proconsortium.org/). Due to the diversity of features by which protein fragments are annotated in this ontology, identifying the right subset of terms has turned out to be challenging, but we produced an initial set of terms that are now in the process of being added to the Reach system's bioresources.

From the MITRE team, we received an updated export of drug-virus relations from the Therapeutics Information Protal which we ingested as a set of observations against the COVID-19 EMMAA model (see https://emmaa.indra.

bio/dashboard/covid19?tab=tests&test_corpus=covid19_mitre_tests). The set of applied tests (i.e., observations) went up from 1,839 to 2,641, and the number of explanatory paths found by EMMAA went up from 1,643 to 2,398. In other words, we now produce explanations for an additional 755 drug-virus relationships.

With the UW team, we made technical specifications for how INDRA/EMMAA can provide annotations back to COSMOS that it can use for enhanced document indexing and retrieval. The two options (each with different advantages) are to (1) integrate additional INDRA processing steps with the reading infrastructure running on xDD and allow COSMOS to ingest these outputs directly or (2) use assembled EMMAA knowledge and map these back (via document identifiers) to COSMOS as annotations. We also discussed approaches to access relevant figures and tables connected to statement evidence. UW will implement an API which takes a set of keywords, and optionally, a set of DOIs and returns a ranked list of figures and tables.

As for the integration with Uncharted, we implemented a new JSON-L format for exporting and sharing EMMAA models and made this available. We also provided ongoing help with accessing and interpreting the content of EMMAA models as well as the results of EMMAA explanations. In support of the latter, we developed a new JSON-L based representation format for tests that provide a list of node names, a list of Statement hashes representing edges, and other metadata necessary to identify the test for which the explanatory path was produced. We also provided an export of all assembled knowledge potentially relevant to any of the EMMAA models, as well as access to a query API for the same knowledge.

### 6.1.4 Open Search model queries and notifications

During this reporting period, we added a new "Open Search" capability to EMMAA's model queries and notifications feature. Until now, EMMAA's notification tools have been focused on identifying new *explanations* for observed cause-effect relationships. The primary use case for this feature is to support scientists who are interested in understanding possible *mechanisms* for a known biological effect.

With Open Search, users can specify a target and get updates on newly discovered regulators of the target (e.g., drugs), or downstream effects (e.g., phenotypes). The motivation for this feature was to allow users to be notified of new discoveries suggesting repurposable drugs for COVID-19. Not only can the user specify the type of target they are searching from (e.g., the disease "COVID-19" or the viral co-receptor protein "TMPRSS2"), but also class of entities they are searching for (e.g., chemicals, proteins, or phenotypes).

The figure below illustrates an EMMAA notification email for a variety of different open searches, including chemicals affecting diseases ("COVID-19"), viruses ("Middle East Respiratory Syndrome Coronavirus") and proteins ("ACE2", "TMPRSS2", "CTSB"). In addition, it includes a search for new downstream effects of a particular drug, "leupeptin":

**Updates to your open queries**

- "What inhibits COVID-19? (CHEBI, DRUGBANK, CHEMBL, PUBCHEM)" in covid19 using the Unsigned Graph model type. For details click here.
- "What inhibits TMPRSS2? (CHEBI, DRUGBANK, CHEMBL, PUBCHEM)" in covid19 using the Signed Graph model type. For details click here.
- "What inhibits TMPRSS2? (CHEBI, DRUGBANK, CHEMBL, PUBCHEM)" in covid19 using the Unsigned Graph model type. For details click here.
- "What inhibits COVID-19? (CHEBI, DRUGBANK, CHEMBL, PUBCHEM)" in covid19 using the Signed Graph model type. For details click here.
- "What inhibits ACE2? (CHEBI, DRUGBANK, CHEMBL, PUBCHEM)" in covid19 using the Signed Graph model type. For details click here.
- "What inhibits ACE2? (CHEBI, DRUGBANK, CHEMBL, PUBCHEM)" in covid19 using the Unsigned Graph model type. For details click here.
- "What inhibits Middle East Respiratory Syndrome Coronavirus? (CHEBI, DRUGBANK, CHEMBL, PUBCHEM)" in covid19 using the Signed Graph model type. For details click here.
- "What inhibits Middle East Respiratory Syndrome Coronavirus? (CHEBI, DRUGBANK, CHEMBL, PUBCHEM)" in covid19 using the Unsigned Graph model type. For details click here.
- "What does leupeptin inhibit? (CHEBI, DRUGBANK, CHEMBL, PUBCHEM)" in covid19 using the Signed Graph model type. For details click here.
- "What does leupeptin inhibit? (CHEBI, DRUGBANK, CHEMBL, PUBCHEM)" in covid19 using the Unsigned Graph model type. For details click here.
- "What inhibits CTSB? (CHEBI, DRUGBANK, CHEMBL, PUBCHEM)" in covid19 using the Unsigned Graph model type. For details click here.

As with notifications for causal paths, EMMAA keeps track of the previously reported results for the query and generates updates for new results. The following image shows the initial set of paths returned for the query "What inhibits COVID-19" in the unsigned network model:

| | |
|---|---|
| losartan → ACE2 → COVID-19 | **losartan → ACE2**<br>Losartan activates ACE2.<br>Losartan increases the amount of ACE2.<br>Losartan deubiquitinates ACE2.<br>**ACE2 → COVID-19**<br>ACE2 inhibits COVID-19.<br>ACE2 binds COVID-19.<br>ACE2 increases the amount of COVID-19.<br>ACE2 activates COVID-19.<br>COVID-19 binds ACE2. |
| pioglitazone → ACE2 → COVID-19 | **pioglitazone → ACE2**<br>Pioglitazone activates ACE2.<br>Pioglitazone increases the amount of ACE2.<br>Pioglitazone decreases the amount of ACE2.<br>Pioglitazone inhibits ACE2.<br>**ACE2 → COVID-19**<br>ACE2 inhibits COVID-19.<br>ACE2 binds COVID-19.<br>ACE2 increases the amount of COVID-19.<br>ACE2 activates COVID-19.<br>COVID-19 binds ACE2. |

The paths show that EMMAA identifies drugs linked to COVID-19 via an intermediate node, the viral receptor ACE2: both of the paths highlighted pointed to ACE2 inhibitors as possibly relevant drugs. While losartan entered clinical trials early on as a potential COVID-19 therapeutic, piaglitazone was discussed only recently as potentially relevant (see the paper "Can pioglitazone be potentially useful therapeutically in treating patients with COVID-19?"). With this initial baseline established, we will be monitoring the results of these open searches for findings with implications for new drug repurposing candidates.

## 6.2 ASKE-E Month 2 Milestone Report

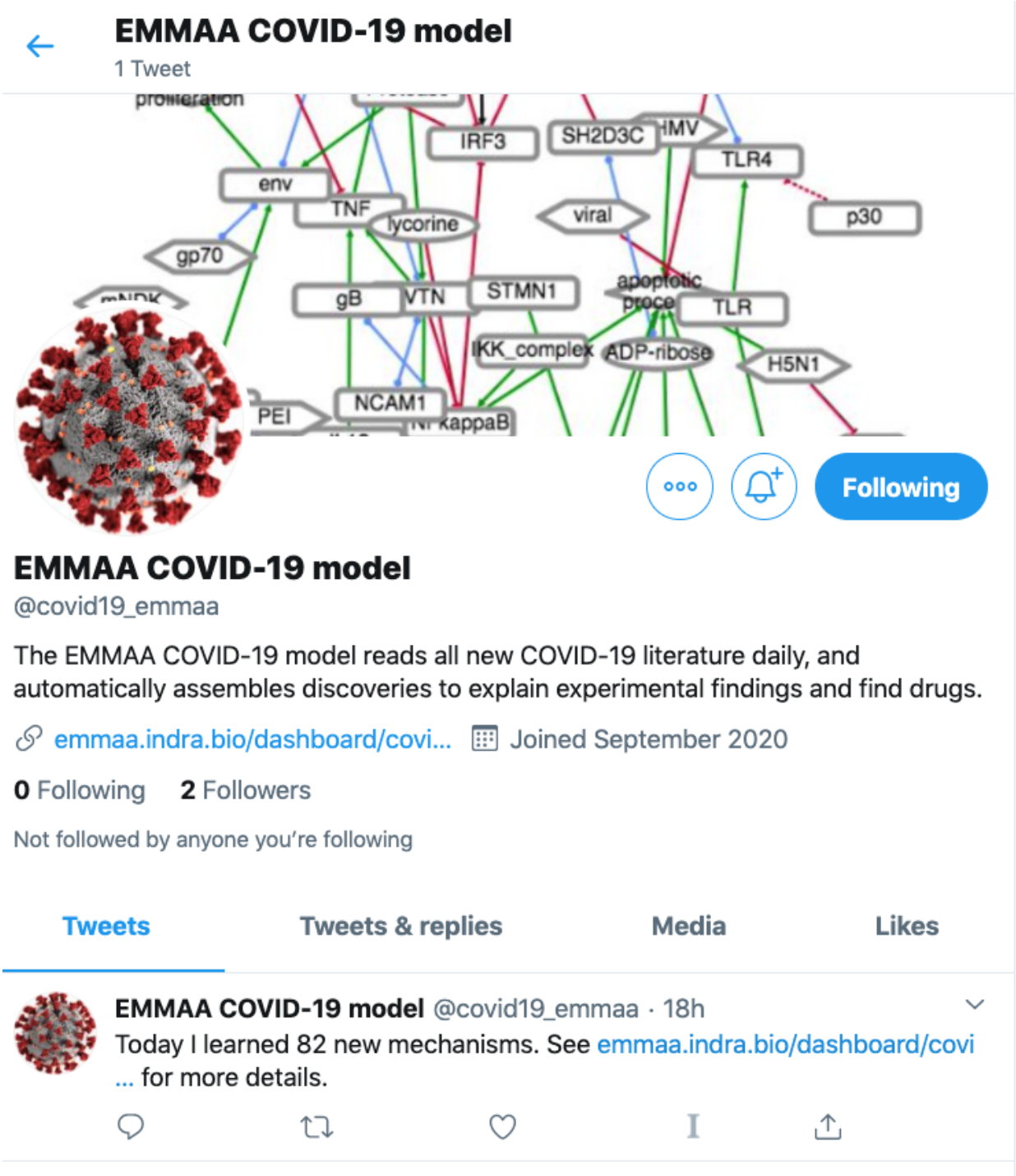
### 6.2.1 Push science: EMMAA models tweet new discoveries and explanations

This month we implemented and deployed Twitter integration for multiple EMMAA models. We have previously developed a proof of concept for Twitter integration, however, that framework had significant limitations. First, tweets only described structural updates to a model (i.e., the number of new statements that were added) and did not report on any functional changes or non-trivial new insights that were gained from the model update. Second, the tweets did not point to any landing page where users could examine the specific changes to the model. In the new Twitter integration framework, we addressed both of these crucial limitations.

Twitter updates are now generated for three distinct types of events triggered by the appearance of new discoveries in the literature:

- New (note that "new" here means that a statement is meaningfully distinct from any other statement that the model previously contained) statements added to a model.
- The model becoming structurally capable to make a set of new explanations with respect to a set of tests (e.g., experimental findings). This typically happens if a new entity is added to the model that was previously not part of it.
- The model explaining a previously unexplained observation (in other words, passing a previously failing "test"). These notifications are particularly important conceptually, since they indicate that the model learned something from the newly ingested literature that changed it such that it could explain something it previously couldn't.

The image below shows the first tweet from the [*EMMAA COVID-19 model*](https://twitter.com/covid19_emmaa).




Crucially, each of the tweets above include a link to a specific landing page where the new results can be examined and curated (in case there are any issues).

Overall, this framework constitutes a new paradigm for scientists to monitor the evolving literature around a given scientific topic. For instance, scientists who follow the EMMAA COVID-19 model Twitter account get targeted updates on specific new pieces of knowledge that were just published that enable new explanations of drug-virus effects.

### 6.2.2 Improving named entity recognition in text mining integrated with EMMAA models

Having evaluated the performance of integrating protein cleavage product names from the Protein Ontology with the Reach reading system's resources, we found that the space of protein fragments covered and the quality of synonyms was insufficient. We therefore implemented an alternative approach that involves extracting protein chain and fragment names from UniProt and using these as synonyms for grounding purposes (see [*Pull request*](https://github.com/clulab/bioresources/pull/42)). We found that this approach adds around 50 thousand new, high-quality lexicalizations for protein fragments, including a large number of human proteins (e.g., Angiotensin-2) and viral proteins (e.g., Nsp1) that are of interest for COVID-19 and many other applications in biology. The UA team is currently working on finalizing these updates and we hope to run an updated version of Reach on the COVID-19 literature next month.

### 6.2.3 Making model tests and paths available for use by other applications

To facilitate integration of EMMAA test results with other applications we made data on model tests and causal paths available for programmatic download. This feature was requested by the Uncharted team, who is exploring approaches to visualize and interact with EMMAA results. The test and path data are stored in public JSON-L files on Amazon S3 and are updated daily. Model test files contain a JSON representation of the EMMAA test statements; test path files list the path nodes, the statement hashes supporting each edge in the path, the hash of the corresponding test, and the type of causal network used to evaluate the test. Downstream applications can get the latest results from each model-test corpus pair from stable Amazon S3 links.

## 6.3 ASKE-E Month 4 Milestone Report

### 6.3.1 EMMAA Neurofibromatosis Models and NF Hackathon Prize

During this reporting period we won one of the top prizes in the "Hack for NF", a six-week event sponsored by the Children's Tumor Foundation to develop novel software relevant to neurofibromatosis (NF), a set of cancer syndromes that affect children.

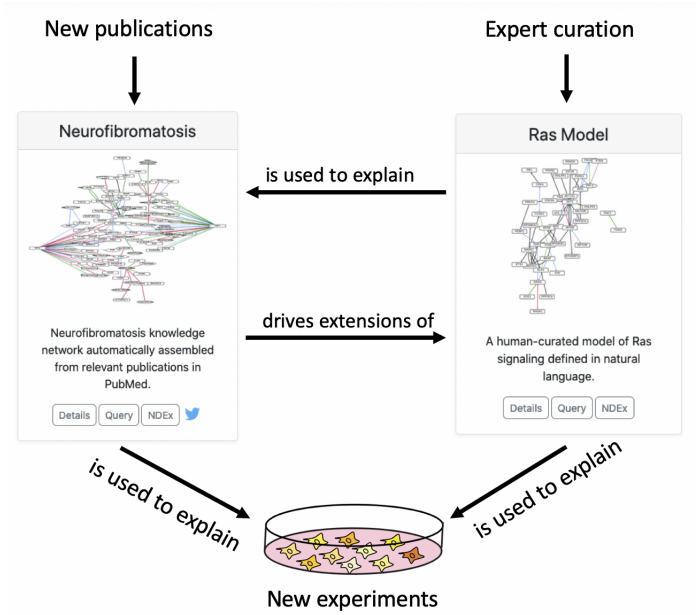
Our submission consisted of two causal models of NF deployed in EMMAA. The first model was built directly from text mining the 18,000 PubMed articles about NF; it contains approximately 9,000 statements about the functions and interactions of NF1, NF2, and other entities mentioned in those articles. Unlike the other cancer-related models in EMMAA, the NF model does not specify an explicit list of disease-relevant proteins: the scope of the model is defined strictly by neurofibromatosis keyword search terms. This keeps the content of the model as disease-specific as possible, with the model serving as a comprehensive representation of what is known about NF.

For the second part of our submission, we substantially expanded our curated Ras signaling model to include mechanisms relevant to NF1 and NF2 signaling. The model is transparent even for non-modelers because it is built from ~200 declarative English sentences and automaticalaly assembled by INDRA. In an iterative, test-driven process, we used the reported causal relationships from the literature-based NF model that were unexplainable by the curated model to both 1) identify errors in the literature derived model and 2) discover necessary extensions to the curated model.

As an example, the literature-based model contains the observation that NF2 inhibits PAK1. The extended curated model shows that this this finding can be explained by a mechanistic path whereby NF2 competes Angiomotin (AMOT) away from inhibiting ARHGAP17, allowing ARHGAP17 to inhibit CDC42, which would otherwise activate PAK1.

As a further demonstration of the scientific value of automated model analysis, we converted drug screening data from NF1 and NF2 cell lines into EMMAA tests and checked the literature-derived model against them. Interestingly, we found that while the causal paths identified by the models were typically short, involving paths with a single intermediate node (i.e., drug->protein->cell proliferation) the explanatory nodes were highly context-specific, in some cases having been previously identified in the literature as therapeutic vulnerabilities for NF cell lines.

We see the two types of models (curated and literature-derived) as working synergistically to explain experimental results and accumulate actionable knowledge, as shown in the diagram below.




For this hackathon entry, we won one of three top prizes. The press release from the Children's Tumor Foundation can be found here, and a video presentation describing our project can be found here.

### 6.3.2 Rapid initialization of EMMAA models from literature for two new diseases

The new Literature Prior module module makes the instantion of EMMAA models based on a subset of the scientific literature straightforward. As input, the class takes a list of PubMed search terms and optionally a list of Medical Subject Headings. It then automatically identifies relevant publications, and collects all statements from text mining that were extracted from these papers. The model is then uploaded to AWS and is available for daily updates and access via the dashboard. We used this method to start two new EMMAA models, for vitiligo and multiple sclerosis.

### 6.3.3 Downloading EMMAA models in alternative formats

The knowledge assembly approach in EMMAA allows exporting each model in multiple different modeling formalisms. In fact, EMMAA internally uses four different modeling formalisms (PySB, PyBEL, signed graph and unsigned graph) for querying and analysis. However, these formats, and other community standards have not been made available to users through the EMMAA dashboard.

We added multiple exports for each model that are generated during each model update (typically daily) and are available through the EMMAA dashboard. Each model has the following export formats available:

- *json.gz*: A gzipped INDRA Statement JSON dump.
- *jsonl*: An uncompressed dump of INDRA Statement JSONs with one statement per line.
- *indranet*: A tabular (tsv) file where each row represents a single binary interaction between two entities. This format is ideal for building networks from an EMMAA model.

Models that support PyBEL analysis provide a *pybel* export. In addition, models that support analysis at the rule-based executable level are exported into the following formats:

- *bngl*: BioNetGen model representation (http://bionetgen.org/)

- *kappa*: Kappa model representation (https://kappalanguage.org/)

Finally, models that support reaction-network based analysis are exported into these formats:

- *sbml*: Systems Biology Markup Language (http://sbml.org/)
- *sbgn*: Systems Biology Graphical Notation (https://sbgn.github.io/)

## 6.4 ASKE-E Month 5 Milestone Report

### 6.4.1 Semantic filters to improve model analysis

Examining the explanations produced by the COVID-19 EMMAA model for in-vitro drug screening experiments, we found that some of the explanations included causal mechanisms that were not consistent with the nature of the experimental context being studied. For instance, in an experiment where a single drug is added in a controlled manner, a mechanism that involves another drug (for instance via a drug-drug interaction) is not appropriate. Similarly, for an in-vitro experiment, higher-level societal factors are semantically not appropriate as intermediate concepts on a causal path.

Motivated by this, we implemented an approach to applying semantic filters to mechanistic paths that allow encoding constraints on what is and isn't allowed on paths when explaining a given observation. These constraints derive from what is known about the experimental context in which that observation was made. For the observations used as test conditions for the COVID-19 model, we created constraints to exclude small molecules other than the drug that is used in the given experiment and higher level concepts including phenotypes, organisms and diseases. We found that the quality of explanations found improved substantially and is now more appropriate semantically with respect to the experimental context.

### 6.4.2 Model analysis exploiting ontological relationships

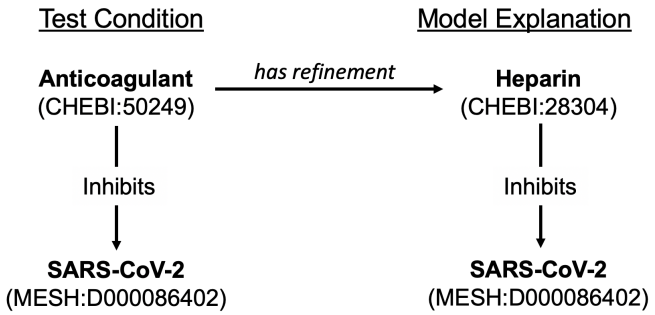
During this reporting period we extended the way EMMAA models are tested against experimental observations. Previously, we applied tests to models based on a strict match between the entities in the test and the set of entities in the model. However we noticed that in many cases models tended to consist of highly specific entities (e.g., individual proteins like KRAS, HRAS, and NRAS), whereas literature mining often picked up tests involving higher-level ontological concepts (e.g., the RAS protein family). The limitation of this approach was that we could only return a path based on exact matches between test and model entities, even when the model contained a path among more specific entities that would serve as a test explanation.

In the new approach we allow relations among more specific concepts to serve as explanations for relations among more general concepts (but not the reverse). Specificity is determined not only by hierarchical levels in the ontology (e.g. a member of a protein family is more specific than the family entity), but also by the amount of contextual information supplied for an entity (e.g., a protein with a phosphorylation is a more specific version of the same entity without a phosphorylation). This information is used to determine which tests can be applied to the model and also to find explanatory paths. To make this relationship explicit in our explanations, when a path found starts or ends with a more specific version of a test entity, we add a special "is a refinement of" or "has a refinement" edge to the path.

We applied this new testing approach to the EMMAA COVID-19 model. For the tests from the MITRE Therapeutic Information Browser Corpus ("MITRE Tests"), 174 new tests were determined to be relevant to the model when taking refinements into account. For these tests, which generally take the form "drug X inhibits virus Y", we found relevant, more specific agents both for drugs (e.g., "rifampicin" is a type of "RNA polymerase inhibitor") and viruses (e.g., "infectious bronchitis virus" is a type of "gammacoronavirus"). Of these new tests, 95 passed in the signed graph network.

An example new passing test is shown in the figure below for the test condition "anticoagulant inhibits SARS-CoV-2", which was previously determined to not be relevant to the model due to the fact that the model did not contain the specific entity for "anticoagulant" (CHEBI:50249). The model contains the information that heparin (CHEBI:28304),

a type of anticoagulant, inhibits SARS-CoV-2, and the system now returns the explanation that "anticoagulant has refinement heparin; heparin inhibits SARS-CoV-2."




### 6.4.3 Improved reading and assembly of protein chains and fragments

Protein chains and fragments are important both for human and viral biology. In ASKE-E month 2, we reported having extended the Reach reading system with lexicalizations of these entities from UniProt and the Protein Ontology (PR). This month, we made a number of extensions to our software stack to propagate these extensions in a useful way.

First, UniProt and PR have a large number of overlapping entries but neither source provides mappings to the other at the level of protein chains (only full protein entries). We developed a semi-automated approach to find and curate these mappings. We used Gilda to find lexical overlaps between the two ontologies and put these as predictions into the Biomappings repository and curation tool. We then curated these mappings to confirm correct ones and remove incorrect ones. These mappings were then propagated into the INDRA Ontology graph to be used for standardization.

Second, we found that the names of protein chains (similar to the names of full proteins) are ambiguous across organisms. This is especially problematic with the large number of viral species and strains that contain protein chains with identical or similar names. Current machine reading systems including Reach typically cannot disambiguate across these choices and produce highly ambiguous groundings for these viral proteins. Therefore, contextual information needs to be brought in externally to decide which organism to prioritize when selecting a grounding produced by Reach. To this end, we implemented an organism prioritization scheme whereby the user (or some external automated process) can supply a ranked list of organism identifiers to represent priority. This list is then used to guide how to the grounding of proteins and protein chains is selected. For example, if a paper is known to describe SARS-CoV-2 and human biology, one can supply an organism priority list including the identifiers of these two organisms to exclude or de-prioritize any spurious groundings from e.g., other viral strains that are irrelevant in the given context. Further, the organisms which a paper describes can be obtained from annotations that are either provided directly with the paper in PubMed or can be obtained using dedicated NLP systems set up for this task e.g, the MTI system.

Going forward, we will re-process the COVID-19 papers with these features in place and expect that the quality of reading, extraction and assembly for virus-host interactions will improve significantly.

### 6.4.4 Bio ontology optimized for visualization

We implemented a custom export of the INDRA BioOntology graph that is optimized for organizing nodes in a UI. The idea is to create top-level groups of entities that correspond to an intuitive category (e.g., human genes/proteins, non-human genes/proteins, small molecules, diseases, etc.). EMMAA models don't contain this information about their entities directly, rather, they are inferred from identifiers assigned to each entity in a given set of name spaces. However, some name spaces contain multiple types of entities (e.g., MESH contains small molecules as well as diseases) and some entity types are distributed across multiple name spaces (e.g., human genes/proteins can be grounded to HGNC, UniProt, FamPlex, etc.). In this custom export, we split some name spaces and merged others to create a more ideal resolution and shared this export with the Uncharted team.

## 6.5 ASKE-E Month 6 Milestone Report

### 6.5.1 Reading and assembly with context-aware organism prioritization

A key challenge in monitoring the COVID-19 literature and modeling the effect of new discoveries is that descriptions of mechanisms span multiple organisms. First, we need to be able to recognize both viral proteins and human (or other mammalian) proteins in text and find possible database identifiers for them. Second, we need to deal with substantial ambiguity in protein naming between viral species.

By default, the Reach reading system's named entity recognition module is configured to tag only human proteins in text. This month, our team developed a script which cross-references UniProt protein synonyms with the NCBI Taxonomy to allow generating customized named entity resources which include protein synonyms for custom subtrees of the Taxonomy. We used this script to generate named entity resources that include all human proteins as well as protein synonyms for all different viral species. We then compiled a custom version of Reach including these resources.

Next, we implemented a new feature in INDRA which allows processing Reach output with context-dependent organism prioritization. For a given paper with a PubMed ID, we can draw on Medical Subject Headings (MeSH) annotations to find out about organisms that are being discussed. For instance, papers about Ebola are (typically) tagged with the MeSH heading D029043 (https://meshb.nlm.nih.gov/record/ui?ui=D029043), and papers about SARS-CoV-2 with MeSH heading D000086402 (https://meshb.nlm.nih.gov/record/ui?ui=D000086402). Once we have a pre-defined or paper-specific list of relevant organisms, we can process Reach output with this order in place to choose the highest priority UniProt entry for each ambiguous entry having been matched.

While our focus here is on coronaviruses (and in particular on SARS-CoV-2), these new capabilities can be applied to studying other types of existing viruses, or monitoring the literature on future emerging viral outbreaks. We have tested the above grounding approach locally but haven't yet re-processed the entire body of literature (~100k papers) underlying the EMMAA COVID-19 model. We plan to do this in the next reporting period.

### 6.5.2 Preparing for the stakeholder meeting

The EMMAA COVID-19 model is considerably large since it is configured to monitor all of the COVID-19 literature without any further restrictions on model scope. Consequently, for more focused (e.g., pathway-specific) studies, it makes sense to start with subsets of this overall knowledge, and demonstrating this type of more focused model-driven analysis is one of the goals at the upcoming stakeholder meeting. To prepare for this, we defined six distinct ways in which our models and REST services can be used to obtain subsets of knowledge on COVID-19 mechanisms, and to extend them using expert knowledge.

First, the EMMAA COVID-19 model can be queried in at least two ways: using a paper-oriented or an entity-oriented approach. In the paper-oriented case, one searches for elements of the EMMAA COVID-19 model that have support from one or more specific publications. In the entity-oriented case, one defines a list of entities of interest, and queries for all model statements that involve one or more of those entities. The advantage of the paper-oriented approach is that one does not need to curate a specific entity list up front, but due to potential recall issues with automated reading, there is no guarantee that a mechanism of interest will have been extracted from any specific paper. In contrast, the entity-oriented approach provides more reliable coverage for the given set of entities while potentially, inadvertently ignoring other relevant mechanisms.

Second, the general INDRA DB can be used to query for information. The REST API supports both entity-oriented and paper-oriented queries here as well. The main difference compared to querying the EMMAA model is that the INDRA DB results are unfiltered (they can statements that have been marked as incorrect, ungrounded entities, statements out of scope, etc.) and may require post-processing to get good quality results for a focused modeling study.

Finally, we provide features for experts to build models from scratch or extend automatically initialized models. For instance, the INDRA API provides an endpoint to run a machine-reading system on a given span of text (e.g., one describing mechanisms for a given pathway in simple English sentences) and process these into INDRA Statements.

We provided pointers to the Uncharted team for invoking all of these service endpoints.

### 6.5.3 Reporting curation statistics

While the update and assembly of EMMAA models is automated, users can manually curate model statements to remove any incorrect extractions and provide better mechanistic explanations. Previously, the EMMAA dashboard allowed submitting and browsing individual curations, but we did not have UI support for users to see statistics on curations at the model level. To address this, we added a new "Curation" tab on the EMMAA model dashboard. In this tab we show the number of curations submitted by individual curators for statements that are part of a given model. We display the counts for both individual evidences and unique assembled statements. This differentiation is important because each assembled statement may be supported by multiple evidences. In addition, curation information affects the assembly process: all statements that have been curated as incorrect and do not have any evidences curated as correct are filtered out from the model.

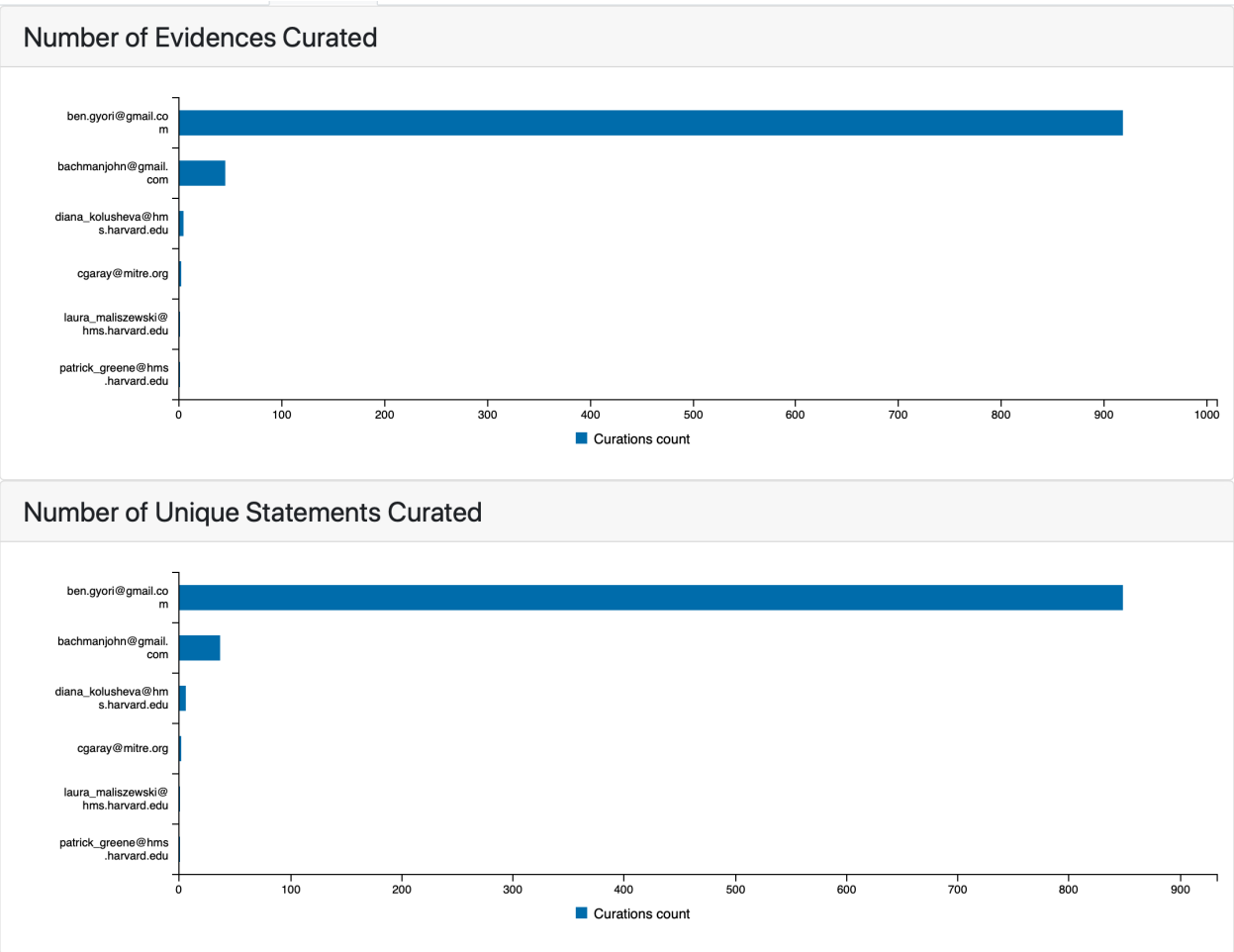



Fig. 1: *Curators of COVID19 EMMAA model*

We also report the number of curations grouped by their type. This shows what errors are the most frequent and helps prioritize further development.

Another aspect of curations we report is how the number of curated statements and evidence changed over time. The figure below shows the time series plot of the number of curations for the COVID-19 model. The first few points here predate the pandemic and the model creation. This is due to the fact the COVID-19 model also integrates a set of older papers on coronaviruses, and some statements from those papers were curated earlier.

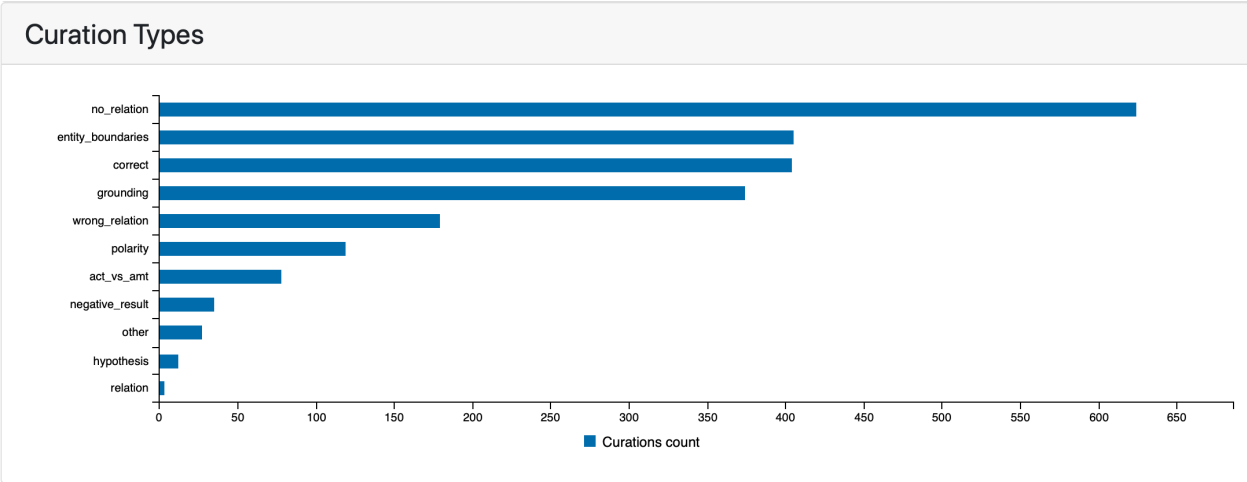



Fig. 2: *Curations grouped by type*

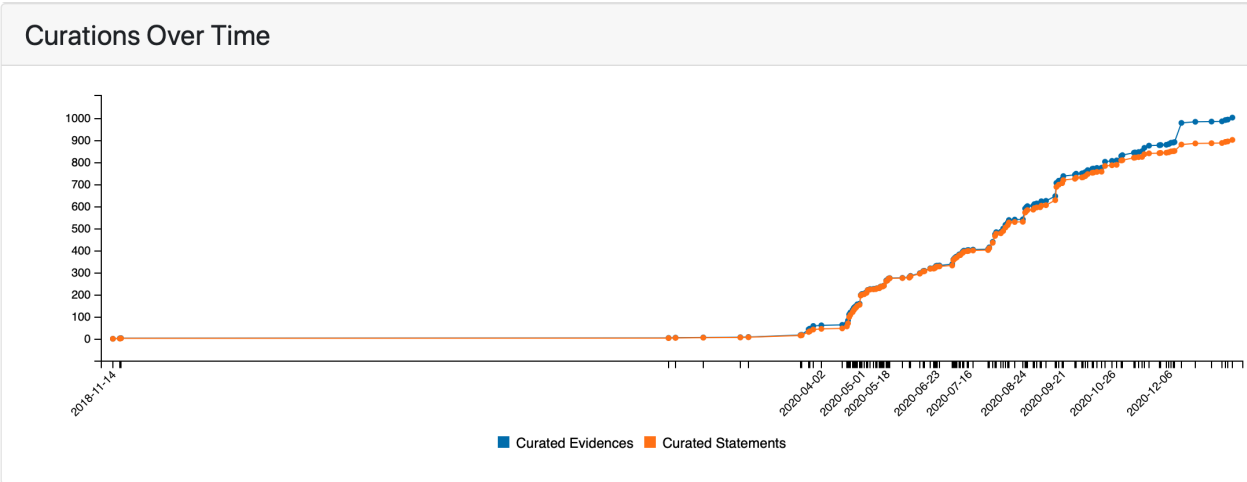



Fig. 3: *Curations over time*

### 6.5.4 Reporting paper level statistics

INDRA processes thousands of publications daily and different EMMAA models make use of different subsets of these. Previously, the EMMAA dashboard didn't provide a dedicated interface for examining the papers that have contributed to each model. In particular, some of the limitations were: 1) It was only possible to see evidences/links to publications for statements that were included in the model after assembly. 2) The evidences/links to publications were grouped by interaction and not by paper. 3) It was not possible to view the papers that produced statements that were filtered out during assembly or papers from which no statements were extracted at all.

In this reporting period we added a new "Papers" tab on each EMMAA model page, and also created a new "statements from paper" service endpoint.

On the "Papers" tab we show the changes in the number of processed papers and the number of papers we get assembled statements from over time.

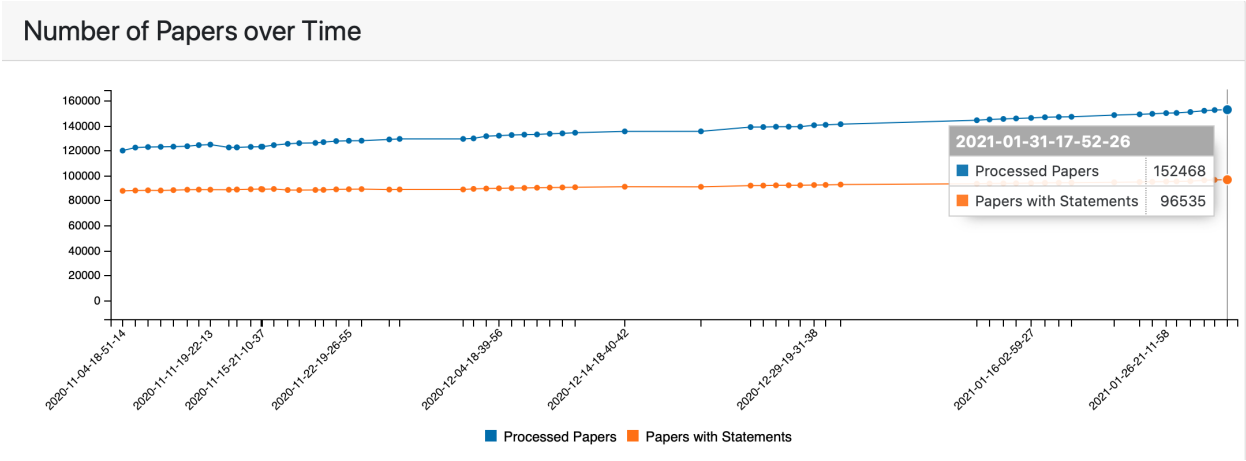



Fig. 4: *Number of processed papers and papers with assembled model statements over time*

We also show the list of papers with the largest number of statements as well as the list of newly processed papers.

New Papers

| Paper Title | Link | Assembled Statements | Raw Statements |
|---|---|---|---|
| Preclinical evaluation of gilteritinib on NPM1-ALK driven Anaplastic Large Cell Lymphoma Cells. | PubMed | 1 | 9 |
| Incidence of Adverse Cutaneous Reactions to Epidermal Growth Factor Receptor Inhibitors in Patients with Non-Small-Cell Lung Cancer. | PubMed | 1 | 2 |
| Epidermal growth factor receptor tyrosine kinase inhibitor remodels tumor microenvironment by upregulating LAG-3 in advanced non-small-cell lung cancer. | PubMed | 1 | 1 |
| Shikonin Inhibits Cholangiocarcinoma Cell Line QBC939 by Regulating Apoptosis, | PubMed | 0 | 16 |

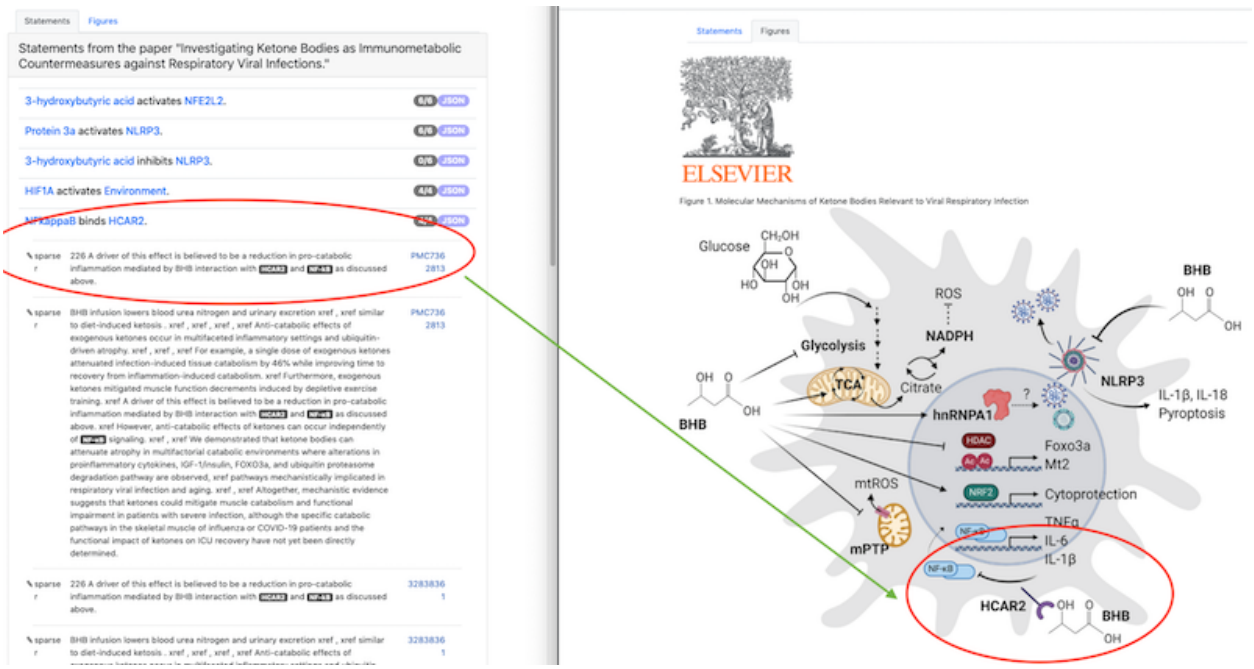
Fig. 5: *Example of new processed papers table*

Each paper title here links out to a new page that shows the model statements extracted from that given paper. This provides a way to explore statements that were all extracted from the same paper. The second column in this table provides a link to the original publication as an external resource.

### 6.5.5 Integrating non-textual evidence with EMMAA models

An important goal in extending EMMAA is to tie the causal mechanisms models are built of to evidence not only in text but also figures and tables. The xDD platform developed at UW provides multiple entry points for querying figures and tables. One approach is to search by entities (e.g., "ACE2, TMPRSS2") to find relevant figures from multiple papers relevant for these entities. Another approach is to search for any figures and tables available for a given paper.

As a proof of principle or integration, we created a client for the second query approach (i.e., find figures and tables by paper identifier) in EMMAA. When displaying the set of statements in an EMMAA model from a given paper, the "Statements" tab allows examining the individual EMMAA statements with their supporting (textual) evidence. A new "Figures" tab contains relevant figures fetched from xDD that can provide additional context and evidence for the model statements.




The figure above shows an initial proof of principle for the paper "Investigating Ketone Bodies as Immunometabolic Countermeasures against Respiratory Viral Infections". On the left, the Statements tab highlights the statement "NFkappaB binds HCAR2" and an evidence sentence describing "…BHB interaction with HCAR2 and Nf-kB…". On the right, the Figures tab shows a directly relevant figure of the interaction between NF-kappaB, HCAR2, and BHB. The visual nature of the figure clearly complements the textual evidence here and may provide users with a richer overall understanding of mechanisms of interest.

This feature is not yet deployed on the main EMMAA dashboard. We are continuing to work on the modes in which figure/table information is integrated with EMMAA and are exploring the possibility of making use of entity-oriented queries to connect figures/tables to EMMAA models.

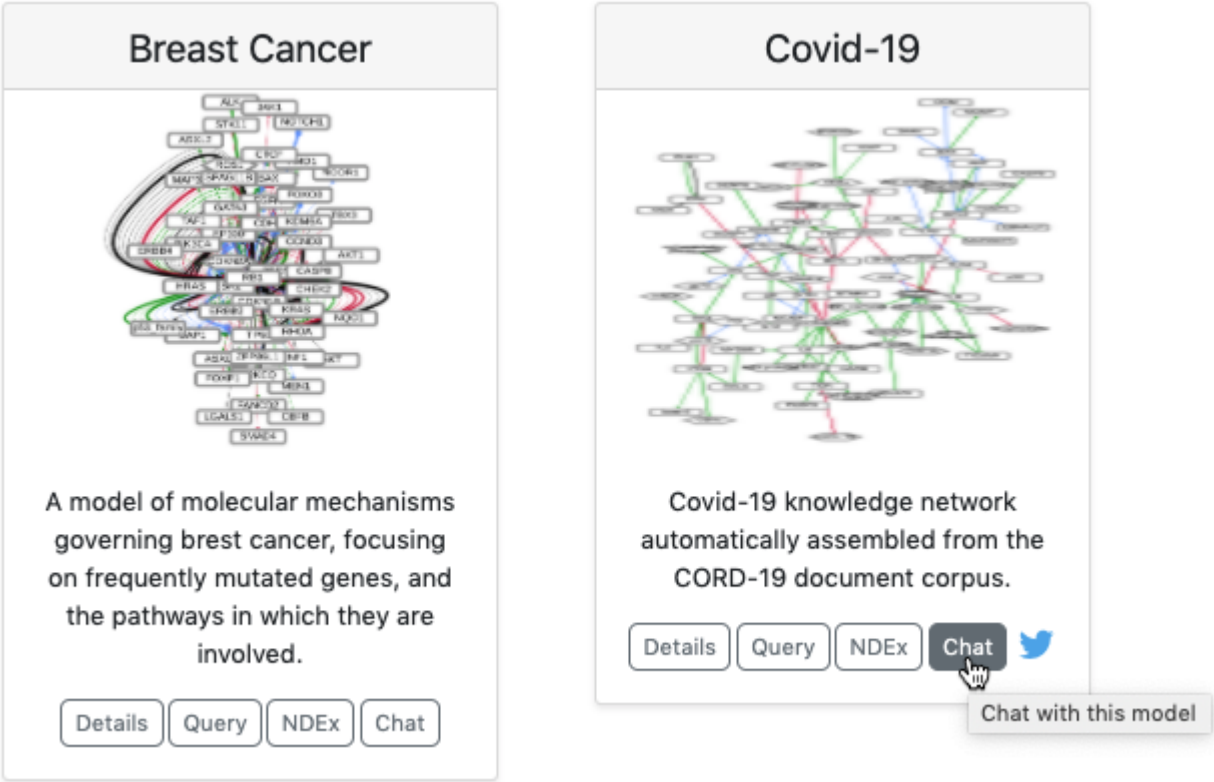
## 6.6 ASKE-E Month 7 Milestone Report

### 6.6.1 Natural language dialogue interaction with EMMAA models

This month we developed a new feature that allows users to directly "chat" with an EMMAA model. The main idea is to make use of the CLARE dialogue system we have previously developed, and create custom instances of it, on demand, that load a given EMMAA model and conduct dialogue with respect to that model. An instance of the

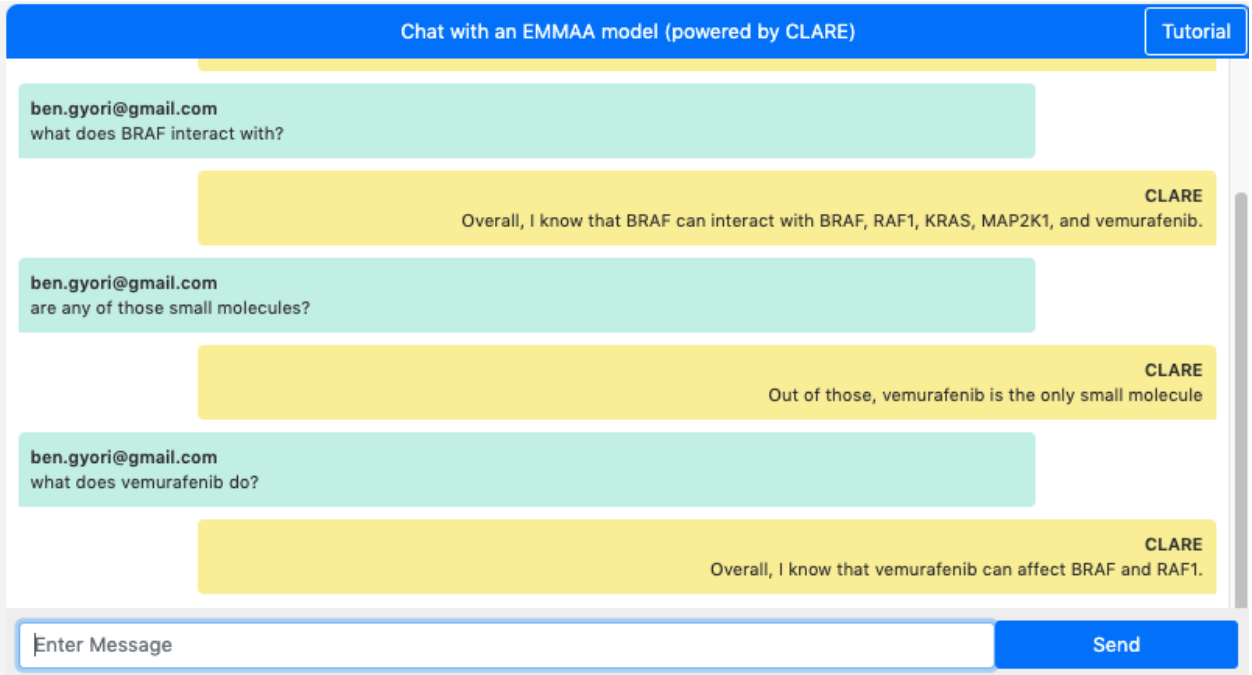
CLARE system is running on a remote server and can handle multiple independent user sessions simultaneously. Chat sessions are orchestrated through the Pusher framework (https://pusher.com/) which handles the real-time aspects of the chat interaction (initialize new user session, asynchronously listen to messages, deliver messages to connected clients, etc.). In EMMAA, we implemented a Pusher chat client which integrates into the main EMMAA dashboard. When clicking on the "Chat" button on the card representing a model on the EMMAA dashboard, a new page opens up where the user can put in their email (this is automatically populated if the user is logged in) and start the chat session. They can then talk about a variety of topics, including mechanisms represented in the given EMMAA model.

The image below shows the new "Chat" buttons on the EMMAA dashboard:

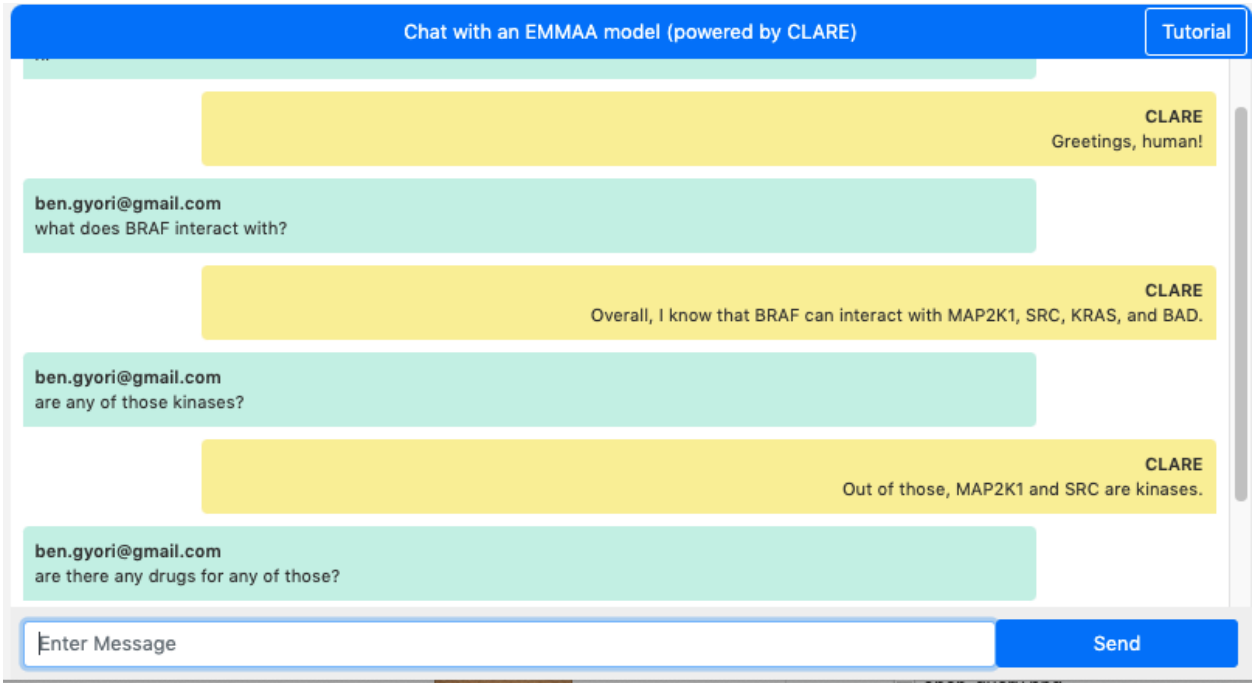



The screenshots below show dialogues with two different EMMAA models: the MARM model above and the RAS model below. The first question "what does BRAF interact with?" highlights the fact that in the two sessions, these questions are answered with respect to two different model contexts. In the MARM model, we find that "BRAF can interact with BRAF, RAF1, KRAS, MAPK1, and vemurafenib", whereas in the RAS model, we find that "BRAF can interact with MAP2K1, SRC, KRAS, and BAD".

Chat with the MARM model:




Chat with the RAS model:




A key feature of human-machine dialogue as implemented by CLARE is that it maintains dialogue context and can interpret and answer follow-up questions using co-references that refer to previous questions or answers. This allows exploring complex mechanisms, such as ones represented by EMMAA models, sequentially. This kind of sequential exploration with intuitive co-reference resolution would be difficult to implement using traditional form-based web interfaces.

The two dialogues above also demonstrate this context-aware co-reference resolution feature. For instance, the questions "are any of those small molecules?", "are any of those kinases?" or "are there any drugs for any of those?" are all questions making use of this feature.

We plan to improve the rendering of some answers (bulleted lists, HTML formatting, etc.) in the coming weeks. We will also improve session management on the back-end to allow terminating sessions explicitly thereby freeing up resources. Finally, we plan to make more tutorials and demos available for this dialogue integration to help users make best use of it.

### 6.6.2 Automatically generated text annotations in context

We implemented a new integration with the hypothes.is that allows taking statements extracted from a given paper, and annotating the website for that paper (a PubMed or PubMed Central landing page, or publisher-specific page) with these statements. First, we implemented an approach to deriving annotation objects from statements. Each hypothes.is annotation consists of a URI (i.e., the address of the page to be annotated), annotation text (i.e., the actual content of the annotation), and a target (a specific text span on the web page that the annotation applies to). The annotation text represets a human-readable English sentence derived from the statement with the names of entities rendered as links to outside ontologies representing them. The target of the annotation is the evidence sentence from which the statement was originally extracted. We can then use the hypothes.is API, for which we implemented a new and extended client, to upload these annotations on demand for a given paper.

We then integrated with new feature with EMMAA. As an extension of the paper-centered view of model statements reported last month, we added a new "hypothes.is button" which allows annotating a given paper on demand and then looking at the annotations in the context of the actual paper. The figure below illustrates the relevant part of the updated "Paper" tab on the EMMAA dashboard.

New Papers

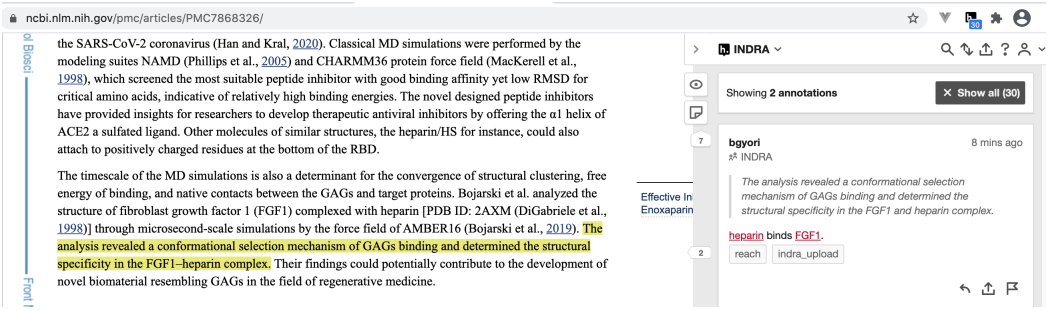
| Paper Title | Link | Assembled Statements | Raw Statements |
|---|---|---|---|
| Focusing on the Cell Type Specific Regulatory Actions of NLRX1. | PubMed h. | 24 | 71 |
| Elucidating the Interactions Between Heparin/Heparan Sulfate and SARS-CoV-2-Related Proteins-An Important Strategy for Developing Novel Therapeutics for the COVID-19 Pandemic. | PubMed h. *Wait while we add annotations* | 23 | 75 |
| Roles of peptidyl-prolyl isomerase Pin1 in disease pathogenesis. | PubMed h. | 16 | 29 |

For each paper from which statements were extracted, a small hypothesis ("h.") badge is now displayed. Clicking on this badge starts the process of uploading the annotations for statements extracted from this paper. After all annotations are added, an external page with this paper opens up in a new tab. In addition, a link to this page is displayed on the EMMAA website.

New Papers

| Paper Title | Link | Assembled Statements | Raw Statements |
|---|---|---|---|
| Focusing on the Cell Type Specific Regulatory Actions of NLRX1. | PubMed h. | 24 | 71 |
| Elucidating the Interactions Between Heparin/Heparan Sulfate and SARS-CoV-2-Related Proteins-An Important Strategy for Developing Novel Therapeutics for the COVID-19 Pandemic. | PubMed h. *Annotations added, see here* | 23 | 75 |
| Roles of peptidyl-prolyl isomerase Pin1 in disease pathogenesis. | PubMed h. | 16 | 29 |

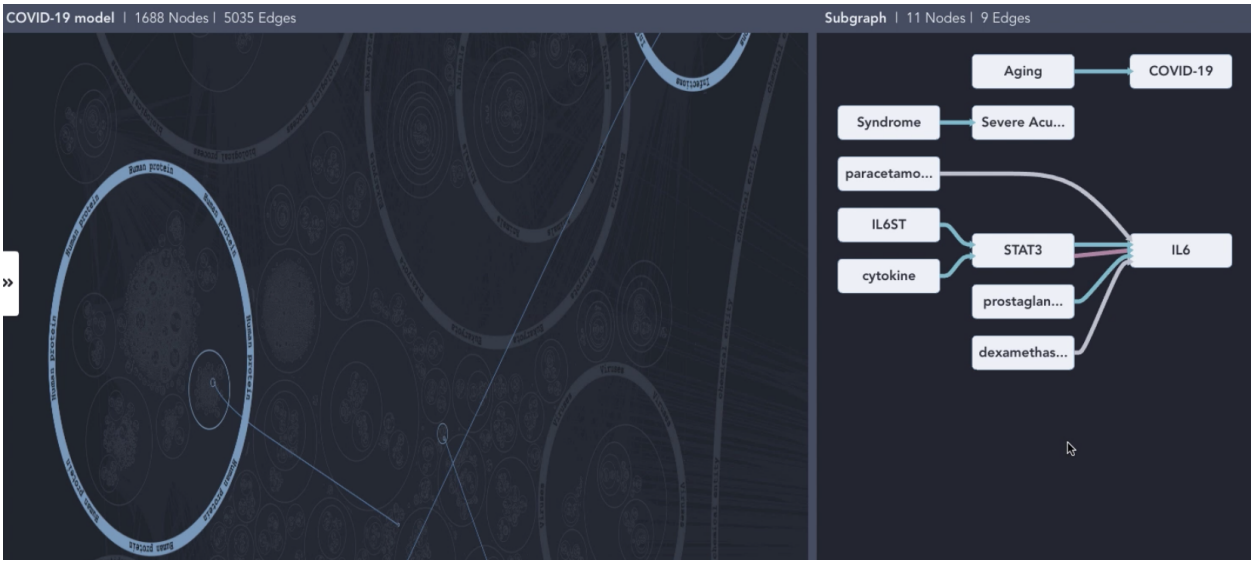
Viewing the uploaded annotations requires the user to install the hypothes.is extension in their browser. The figure below shows how annotations can be viewed and edited on the newly opened page. In this example, a paper on PubMed Central was automatically annotated. The sentences supporting each of the extracted statements are highlighted in the paper and the statements can be viewed in the annotations panel on the right. For instance, this image shows the highlighted sentence mentioning "FGF1–heparin complex" and the extracted "heparin binds FGF1" INDRA statement.




Currently, these annotations are only visible by members of a closed group on hypothes.is, however, we have requested that hypothes.is make annotations in the group publicly visible, and hope that this will be done soon.

### 6.6.3 Demonstrations at the stakeholder meeting

The February 2021 stakeholder meeting focused on system integration: we demonstrated how EMMAA models can be displayed and interacted with in the HMI developed by Uncharted. First, we showed how a keyword search for an entity of interest can lead a user to "discover" a relevant paper and then an EMMAA model which contains mechanisms surrounding the given entity. The user can then interact with a network view of the model, highlighting interactions derived from the paper of interst in the context of all concepts organized by their ontological categories (for instance, a search for IL6 connects the node representing it in the "Human proteins" category with the node representing SARS in the "Infections" category). The HMI is also able to visualize the subnetwork corresponding to the specific paper on a separate tab. The user can then click on a node to see additional incoming or outgoing interactions and click ont them to add them to this view. The figure below shows interactions highlighted in the context of ontology-based categories on the left, and the separate view of interactions derived from a given paper on the right.




We also showed how the results of model queries can be displayed in the HMI. Here we focused on small molecules that can inhibit the replication of SARS-CoV-2 through an intermediary of interest: the Nrf-2 (NFE2L2) protein. Based on the ontology-guided grouping, the HMI provides an intuitive overview of what types of entties are on each mechanistic path from a drug to SARS-CoV-2. For instance, sildenafil, which is grouped under "vasodilator agents" is shown to regulate the activity of NFE2L2 which in turn can regulate SARS-CoV-2 replication. We also showed examples of drugs inhibiting SARS-CoV-2 via cathepsins. The figure below shows mechanisms by which drugs regulate SARS-CoV-2 via NFE2L2. More detail can be seen by zooming and panning in the HMI.

### 6.6.4 Developing the EMMAA REST API for flexible integration

We continued working on extending the EMMAA REST API to support integration with other teams. One of the key goals was to allow dynamic retrieval of EMMAA models and tests metadata. To enable this, we implemented four new endpoints in the EMMAA REST API that support the retrieval of the following data:

- A list of all available EMMAA models;
- Model metadata (short name, human readable name, description, links to the NDEx landing page and to the model's Twitter account) for a given model;
- A list of test corpora that a given model is tested against;
- Test corpus metadata (name and description) for a given test corpus.

Another important extension of the EMMAA API we implemented is the support for running queries programmatically. Previously it was only possible to submit queries through a web form on the Query page of the EMMAA dashboard and then browse the displayed results. The new approach allows sending programmatic requests to the API and receive the results in JSON format. Similar to the interactive interface on the dashboard, the programmatic endpoint supports three types of queries: static (find directed paths between two entities), open search (find upstream regulators or downstream targets of an entity), and dynamic (confirm dynamical model properties by simulating the model) queries.

## 6.7 ASKE-E Month 9 Milestone Report

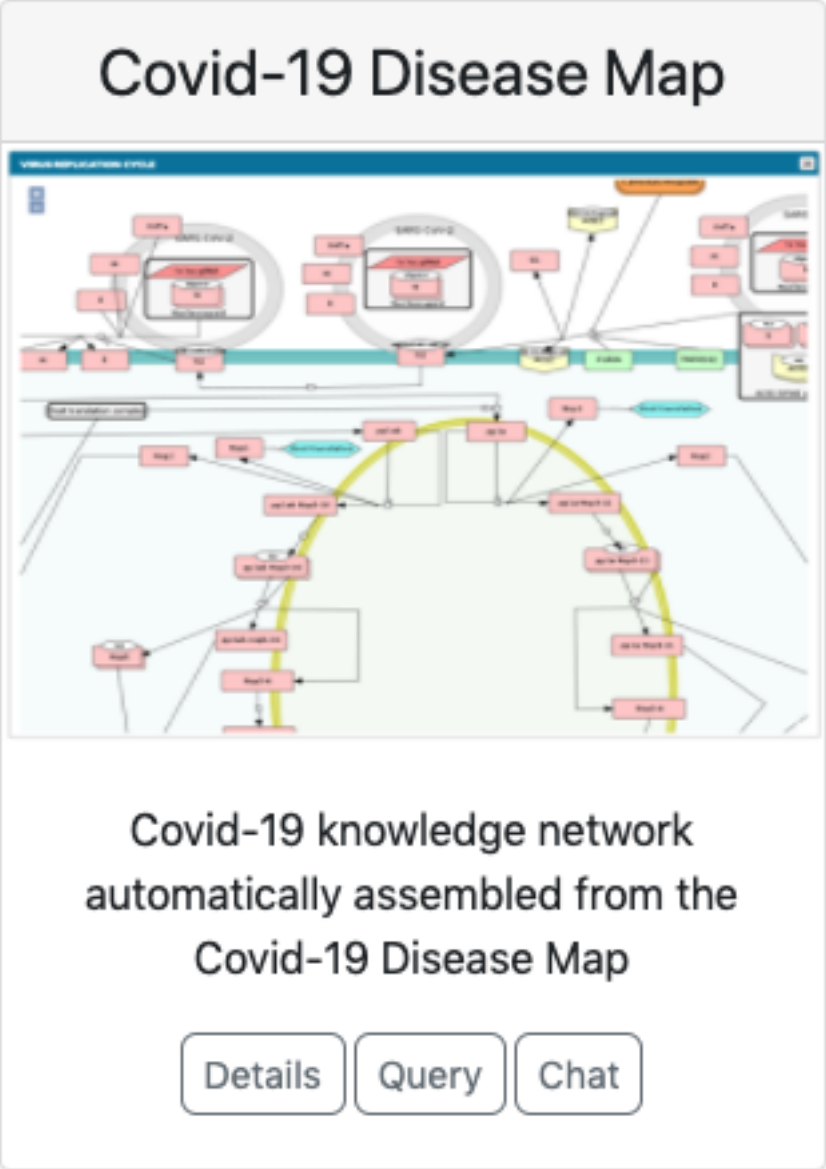
### 6.7.1 Integrating the COVID-19 Disease Map community model

One of our goals in this project is to demonstrate the capability to take an existing model constructed by others in the community and instantiate it as an EMMAA model. One approach for doing this is to take the model in its original form and extend it with some meta-data to allow running it for the purposes of validation and analysis within EMMAA. Another approach is to process the original model into knowledge-level assertions - in our case INDRA Statements - and instantiate this set of statements as an EMMAA model. As the first proof of principle, we decided to take the latter approach since it results in a more transparent model with all necessary annotations available to display model statistics, testing and query results on the EMMAA dashboard. Due to its direct relevance to our applications and its interesting connection with our existing automatically assembled COVID-19 EMMAA model, we decided to work with the COVID-19 Disease Map model.

The COVID-19 Disease Map (C19DM) is a large model of molecular mechanisms related to SARS-CoV-2 infection and COVID-19 curated collaboratively by a consortium of experts. It models all known SARS-CoV-2 protein interactions with human host proteins, and multiple pathways that are triggered by these interactions. It also models phenotypic outcomes associated with COVID-19, for instance, cytokine storm, thrombosis, vascular inflammation, ARDS, etc.

The C19DM is being built using CellDesigner and can be explored or programmatically obtained through the MINERVA platform. Using the CASQ tool, the model has also been transformed into a Simple Interaction Format (SIF) that can be used as the basis for causal analysis or Boolean/logical modeling.

We implemented a new client and processor in INDRA to process the C19DM SIF files in conjunction with meta-data (entity grounding, literature references, etc.) from MINERVA into INDRA Statements. We then initialized an EMMAA model with these statements (see model card below).




The next step was to set up the model for automated analysis against a set of relevant empirical observations. We chose three sets of observations to analyze the model against: (1) a set of in-vitro drug screening experiments, (2) a set of empirical assertions on drugs inhibiting SARS-CoV-2 infection or adverse outcomes associated with COVID-19 (aka the MITRE test corpus), and (3) text mining statements automatically collected by the existing EMMAA COVID-19 model. We used both signed graph and unsigned graph instantiations of the C19DM EMMAA model for analysis.

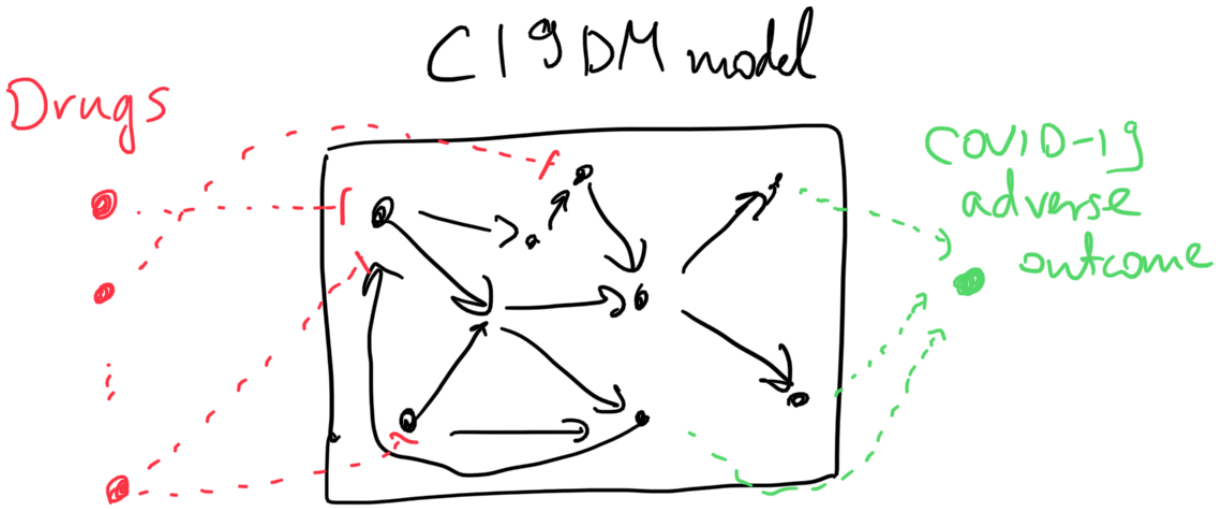
In case of the in-vitro drug screening data, each data point can be interpreted as "Drug X inhibits SARS-CoV-2 replication". Viral replication appears as a concept in the C19DM model and can be used as a readout in this case. However, most drugs that appear in the screening data aren't modeled in the C19DM. We therefore extended the EMMAA C19DM model with these drugs and their known targets within the C19DM using statements independently assembled by INDRA from multiple sources. The in-vitro drug screening corpus is relatively small and the EMMAA C19DM model (instantiated as a signed graph model) could explain 10 such observations, including how "nafamostat inhibits viral replication". The explanation in this case involved nafamostat inhibiting TMPRSS2's activation of ACE2 which enables viral replication:

Test: "Nafamostat inhibits Virus Replication." for COVID19_MAP (Signed Graph) on 2021-04-29

| Path | Support |
|---|---|
| nafamostat ⊣ TMPRSS2 → ACE2 → Virus Replication | **nafamostat ⊣ TMPRSS2**<br>Nafamostat inhibits TMPRSS2.<br>**TMPRSS2 → ACE2**<br>TMPRSS2 activates ACE2.<br>**ACE2 → Virus Replication**<br>ACE2 activates Virus Replication. |

As for the second corpus, these empirical assertions aren't necessarily associated with SARS-CoV-2 replication per se, rather, they imply that a given drug is beneficial in inhibiting some COVID-19-associated adverse outcome. While there are several phenotypic nodes that could be considered readouts for this purpose in the C19DM (e.g., thrombosis, cytokine storm, ARDS, etc.), we don't want to assert up front which one of these a given drug affects. Therefore, we added a new readout concept to the model called "COVID-19 adverse outcome" and added positive regulation relations between each specific adverse outcome concept and this new one. Similar to the case of in-vitro drug screening as described above, we also added external drug-target statements relevant for this corpus.

The sketch below illustrates the two types of model extensions done to make the model applicable to these analysis tasks.




For this corpus the C19DM EMMAA model, instantiated as a signed graph was able to explain 463 test statements. It is particularly interesting to observe which specific phenotypic outcome the explanation involves as a "COVID-19 adverse outcome". For example, for the observation that "aliskiren inhibits COVID-19 adverse outcomes", EMMAA finds an explanation in the C19DM in which aliskiren inhibits angiotensin which - through some intermediaries - leads to reduced vascular inflammation, one of the adverse outcomes associated with COVID-19:

Test: "Aliskiren inhibits COVID-19 adverse outcomes." for COVID19_MAP (Signed Graph) on 2021-04-29

| Path | Support |
|---|---|
| aliskiren ⊣ AGT → angiotensin I → aldosterone → vascular inflammation → COVID-19 adverse outcomes | **aliskiren ⊣ AGT**<br>Aliskiren inhibits AGT.<br>**AGT → angiotensin I**<br>AGT activates angiotensin I.<br>**angiotensin I → aldosterone**<br>Angiotensin I activates aldosterone.<br>**aldosterone → vascular inflammation**<br>Aldosterone activates vascular inflammation.<br>**vascular inflammation → COVID-19 adverse outcomes**<br>Vascular inflammation activates COVID-19 adverse outcomes. |

Finally, we set up the existing EMMAA COVID-19 model (which aggregates knowledge about COVID-19 largely via text mining the existing and emerging literature) as a set of assertions to be explained by the C19DM. Conceptually this is an interesting analysis task since the EMMAA COVID-19 model contains many assertions about indirect effects (e.g., "azythromycin activates autophagy", see below) that were reported in the literature but not necessarily explained mechanistically, while the C19DM is a detailed, mechanistic and high-precision (in that it is human curated rather than automatically assembled) model that is likely to contain mechanistic paths that can serve as interesting explanations. As an example, below is the explanation constructed for the "azythromycin activates autophagy" example.

Test: "Azithromycin activates autophagy." for COVID19_MAP (Signed Graph) on 2021-04-29

| Path | Support |
|---|---|
| azithromycin → TLR3 → TICAM1 → RIPK1 → MAP3K7 → MAP2K4 → JNK → BCL2 → autophagy | **azithromycin → TLR3**<br>Azithromycin activates TLR3.<br>**TLR3 → TICAM1**<br>TLR3 activates TICAM1.<br>**TICAM1 → RIPK1**<br>TICAM1 activates RIPK1.<br>**RIPK1 → MAP3K7**<br>RIPK1 activates MAP3K7.<br>**MAP3K7 → MAP2K4**<br>MAP3K7 activates MAP2K4.<br>**MAP2K4 → JNK**<br>MAP2K4 activates JNK.<br>**JNK → BCL2**<br>JNK activates BCL2.<br>**BCL2 → autophagy**<br>BCL2 activates autophagy. |

Another example involves the observation that "viral N protein downregulates interferon" which the EMMAA C19DM model explains through the SARS-CoV-2 N protein's inhibition of human IRF3.

Test: "N decreases the amount of Interferon." for COVID19_MAP (Signed Graph) on 2021-04-29

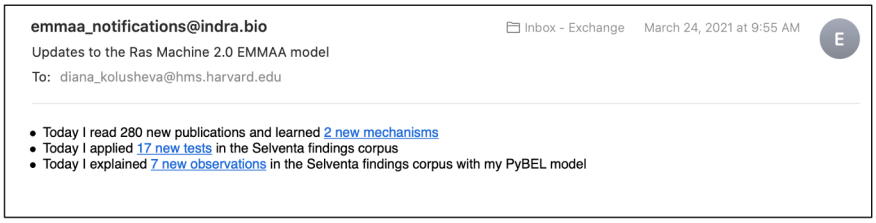
| Path | Support |
|---|---|
| N ⊣ IRF3 → IFNA1 → Interferon | **N ⊣ IRF3**<br>N inhibits IRF3.<br>**IRF3 → IFNA1**<br>IRF3 activates IFNA1.<br>**IFNA1 → Interferon**<br>IFNA1 is a refinement of Interferon. |

Going forward, we will work on instantiating the C19DM as a simulatable Boolean network within EMMAA and will also work towards importing other existing models into EMMAA for automated analysis.

### 6.7.2 Notifications about general model updates

One of the key concepts of EMMAA is "push science" - notifying users of new discoveries relevant to their research. Previously reported developments in this direction included subscribing to query results and tweeting about new findings. We recently made a new step towards this goal and added a feature allowing users to subscribe to a model of their interest.

To subscribe to model notifications, a user needs to click the "Subscribe" button on the model dashboard. The models are updated and tested daily and every time there are any new findings, a subscribed user will receive an email with updates. New findings can include new mechanisms added to the model from the literature, new tests applied to a model, or new explanations found for the tested observations.



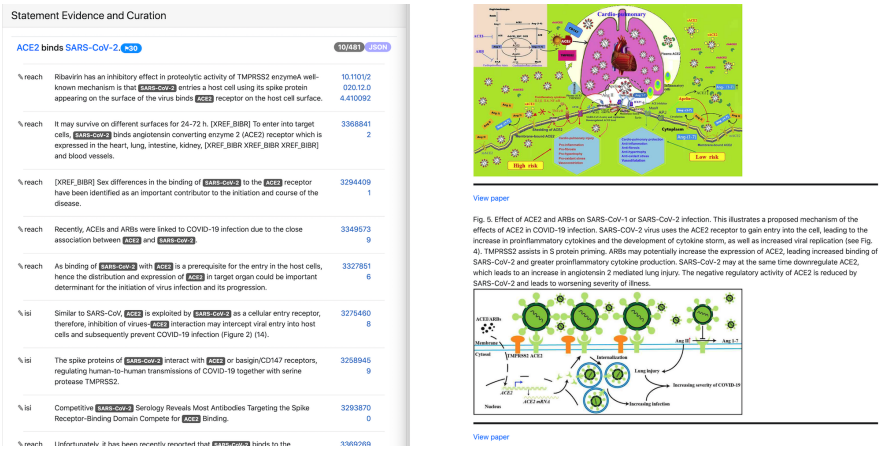

We refactored our code base to separate all code related to notifications (tweets and emails about model updates and emails about new query results) into a *subscription.notifications* submodule. This allows sharing and reusing relevant parts of code.

### 6.7.3 Figures and tables from xDD as non-textual evidence for model statements

We previously reported on displaying figures and tables from a given paper through the integration with the xDD platform developed by UW. That approach supports an exploration of different mechanisms described in the context of a single paper by viewing both their text description and visual representation.

In this reporting period we added support for displaying figures and tables relevant for a given mechanism rather than for a particular paper. To enable this we used xDD entity based search mode that allows searching for objects associated with one or more entities across their knowledge base. For our use case we are searching for figures and

tables where both statement subject and object are involved. As a result, we can display both textual and non-textual evidence for a given statement coming from different papers.




In the image above the text evidence and figures for the statement "ACE2 binds SARS-CoV-2" are shown. Both text and figures are from different papers and have links to the original publications.

### 6.7.4 Integration with the Uncharted UI

We continued working on the integration of EMMAA with the Uncharted UI and made progress on several fronts. Model exploration in the UI is divided into two parts, a large-scale network overview, and a more focused drill-down view.

For the network overview, our concept was to use the INDRA ontology - which is assembled from third-party ontologies in a standardized form - to hierarchically organize nodes in the network (each node represents a biological entity or concept) into clusters. This visualization is most effective and clear if the hierarchical structure of the ontology is fully defined, i.e., every entity is organized into an appropriate cluster, and the hierarchy is organized into an appropriate number of levels. Motivated by this, we spent considerable effort on improving the INDRA ontology's inherent structure, as well as creating a custom export script which makes further changes to the ontology graph specifically to improve the visual layout in the UI.

We also added multiple new features to the EMMAA REST API to support UI integration. For example, we added an endpoint to load all curations for a given model, categorizing curated statement into correct, incorrect and partial labels. Another important feature is providing general information about entities in each model, including a description, and links to outside resources describing the entity. To this end, we implemented a new service called Biolookup (which will be separately deployed) that provides such information for terms across a large number of ontologies in a standardized form. We then added an endpoint in the EMMAA REST API which uses Biolookup to get general entity information and can also add model-specific entity information to the response.

Our teams have also been involved in many ongoing discussions. These included deciding on use cases, visual styles, and all aspects of the interpretation of EMMAA models in order to present them to users in an appropriate way.

### 6.7.5 Semantic separation of model sources for analysis and reporting

When creating a model of a specific disease or pathway, it often makes sense to add a set of "external" statements to the model to make it applicable to a specific data set. A typical example is adding a set of drug-target statements or a set of phenotypic "readout" statements to a model to connect it to a data set of drug-phenotype effects. These external statements should ideally not appear in model statistics. For example, for the COVID-19 Disease Map model, we marked all drug-target and penotype-readout statements as external since these were not part of the original model.

Another categorization of statements in models is "curated" vs "text mined". For instance, the COVID-19 model combines statements mined from the literature with statements coming from curated sources such as CTD or DrugBank. Given that we use the COVID-19 Disease Map Model to automatically explain observations that appear in the COVID-19 Model, it makes sense to restrict these explanations to statements that aren't "curated".

To achieve this, we extended the EmmaaStatement representation to contain metadata on each statement that then allows the statements to be triaged during statistics generation and model analysis.

### 6.7.6 Assembling and analyzing dynamical models

During this period, we aimed to strengthen EMMAA's capability to execute and analyze dynamical models. Previously, EMMAA's dynamical queries supported checking "unconditional" properties, for instance, whether in a model "phosphorylated BRAF is ever high". This captures a model's baseline dynamical behavior without any specific perturbation condition. Further, EMMAA only supported deterministic and continuoys ODE-based simulation of models.

We added support for a new simulation mode, namely continuous-time, discrete-space stochastic simulation using the Kappa framework. One important advantage of this approach is that - unlike the ODE-based approach - it does not rely on enumerating all molecular species that can exist in the system ahead of simulation. Instead, an initial mixture of molecular species is evolved, through a set of reaction rules, and new species can be created during simulation if any reaction rules produce them. However, stochastic simulation is typically slower than ODE-based simulation.

Further, we also implemented a new query mode for dynamical models that can be used to observe model behavior under perturbations. For instance, it allows answering the query "does EGF increase phosphorylated ERK?" in a model by setting up a pair of simulation experiments in which EGF is either at a low or a high level, and then quantifying the difference in the temporal profile of phosphorylated ERK between the two condition (the outcome is either "increase", "decrease" or "no change"). This is useful for interactive user-driven queries but can also be used for model testing/validation against a specific set of observations.

There are numerous challenges involved in evaluating the dynamics of automatically assembled EMMAA models. For very large models such as the COVID-19 model, it makes sense to think of "executable subnetworks" that are assembled to answer a specific set of queries instead of attempting to simulate the entire model. We began implementing an assembly pipeline that performs additional filtering, reasoning and processing on assembled knowledge to prepare if for execution. These steps involve filtering to "direct" statements to remove indirect/bypass effects, rewriting molecular states in statements to improve the causal connectivity of the model, and filtering out "inconsequential" statements to cut down on the size of the model. We also implemented a new analysis feature that can detect potential polymerization (where molecular species can form arbitrarily large complexes as the system evolves) in a model which precludes ODE-based simulation and can result in slower stochastic simulation. For now, these detected polymerizations can help manually patch models, however, it might be possible to automate the addition of constraints to a model to avoid polymerization. Another problem is that of model parameterization. EMMAA models could be connected to relevant expression profiles to set total protein amounts as initial conditions, while reasonable priors can be chosen for reaction rate constants. Beyond that, the uncertainty in model parameters can be resolved by any combination of (1) fitting the model to data, (2) performing ensemble analysis that "integrates" over the model uncertainty, and (3) user interaction to set parameter values manually.

### 6.7.7 Creating a training corpus for identifying causal precedence in text

One of our goals during this period (in collaboration with the UA team) was to extend the Reach reading system with the ability to recognize causal precedence in text. An example of causal precedence expressed in text is the following sentence: "insulin binding of the insulin receptor (IR) at the cell surface activates IRS-1 intracellularly, which in turn activates PI3K". This sentence not only implies that (a) IR activates IRS-1 and (b) IRS-1 activates PI3K but also speficically suggests that (a) is a causal precedent of (b). Given that not all A->B and B->C relationships that are independently collected necessarily imply A->B->C in any specific context, explicit descriptions of such knowledge are extremely valuable for understanding complex causal systems.

One challenge is collecting a large corpus of training data which consists of sentences with causal precedences descrbing some A->B->C causal chain without manual curation effort. Our idea was to start from curated databases to identify causal A->B->C sequences. Knowledge bases such as Reactome, KEGG and SIGNOR are organized into pathways, and the same molecular entity may appear in multiple pathways and be involved in different interaction in each pathway. This implies that to find relevant causal precedence examples, it makes sense to search for A->B and B->C relationships within the scope/context of a single curated pathway (instead of all curated knowledge combined). We ran this search on both Reactome and SIGNOR pathways and found that results from SIGNOR were higher quality and consistent with expected positive and negative controls.

Next, we searched all existing outputs from Reach to find instances of A->B and B->C relationships (from the set identified from SIGNOR) extracted from a single paper, and either a single sentence or two neighboring sentences. We found a total of 782 such sentences automatically. These sentences will become the training set for learning to recognize causal precedence.

We made our code available at https://github.com/indralab/causal_precedence_training and will continue to extend it to find further opportunities for automated training data collection.

### 6.7.8 Knowledge/model curation using BEL annotations

We have previously described an integration with hypothes.is. This integration has supported two usage modes: (1) users can select sentences on any website and add annotations in simple English language that can be processed into statements automatically, and (2) text mined statements can be exported and uploaded as annotations onto the websites (for instance PubMedCentral) where they were originally extracted from.

Though usage mode (1) is convenient, NLP on even simple sentences can sometimes be unreliable and therefore we decided to implement support other intuitive but formal syntaxes for annotation. Our preferred choice was the Biological Expression Language (BEL) which allows expressing a wide range of causal relationships relevant for biology. For instance, the BEL statement "kin(p(FPLX:MEK)) => kin(p(FPLX:ERK))" expresses that the kinase activity of the protein family MEK directly increases the kinase activity of the protein family ERK. Building on the PyBEL package and the existing BEL-INDRA integration we added support for parsing BEL statements from hypothes.is annotations into INDRA Statements. We plan to use this capability to build new human-curated models or extend existing ones in EMMAA.

### 6.7.9 Formalizing EMMAA model configuration

Each EMMAA model has to be set up with its own configuration settings in a JSON file. The settings allow to store model specific metadata (e.g. short and human readable name, links to NDEx visualization and Twitter accounts) that are displayed on the model dashboard as well as to configure the methods to update and assemble the model, run test and queries and generate statistics reports. With the number and diversity of EMMAA models growing we felt the need to document the requirements to the model configuration. The detailed instruction on what information the configuration file should contain with examples can be found at *Configuring an EMMAA model*

## 6.8 ASKE-E Month 10 Milestone Report

### 6.8.1 Dynamical model analysis

We made several developments that significantly extend the ways in which EMMAA models can be analyzed using simulation.

### Extended automated assembly for model simulation

As described previously, EMMAA contains models built using several different approaches ranging from small human-defined models written in simple English to large fully automatically assembled models from up to hundreds of thousands of publications. There is a special set of challenges associated with models built automatically from source knowledge as follows:

- Several EMMAA models are very large, making simulation impractical.
- EMMAA models that are automatically assembled from literature and pathway databases can by default include "bypass edges", i.e., relationships that are reported in some source which are not direct physical interactions but indirect effects.
- There are complicated redundancies at the level of individual mechanisms, for instance a model can simultaneously contain "A activates B", and "A phosphorylates B", without an explicit relationship between the two. This can create inconsistent "parallel" pathways over different states of B.
- Models that include text mining output are naturally subject to some amount of incorrect information due to various random and systematic errors.
- Any mechanisms not explicitly stated in text (or in pathway databases) are not represented. One common set of mechanisms are "reverse effects". For instance, there may be several known mechanisms for the positive regulation of the amount of a given protein, but no explicit mention of the protein naturally degrading.

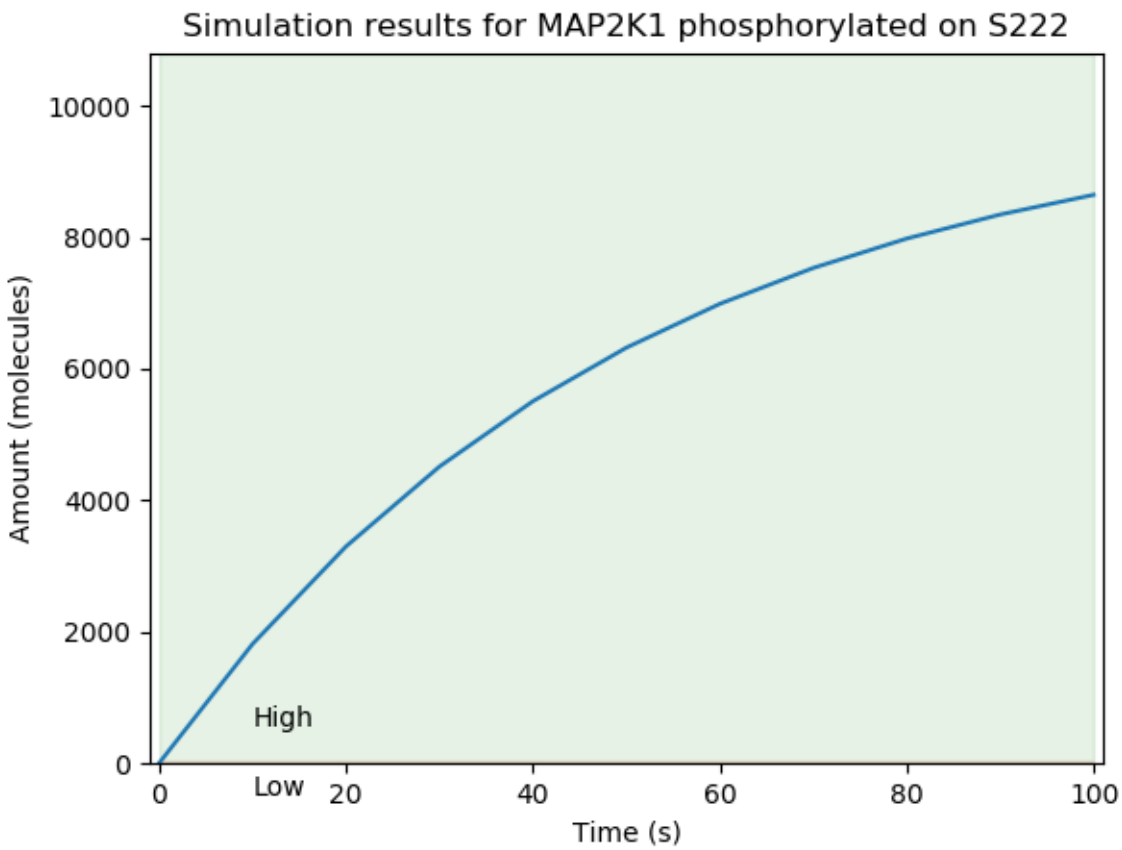
To address these challenges, we have developed a number of assembly procedures and implemented support for running an additional assembly pipeline consisting of these steps for EMMAA models, specifically to support dynamical simulation. Similar to the generic knowledge assembly pipeline that is applied to each EMMAA model, these assembly steps are still applied at the knowledge/statement level before generating a rule-base dynamical model from the statements using the PySB model assembler.

To demonstrate this, we chose The Ras Machine model and configured an extended assembly pipeline with the following steps:

- Filter out complex formation statements, since they can lead to unconstrained polymerization unless additional conditions are supplied.
- Filter to statements that are known to be direct, either based on annotations from pathway databases or determined from linguistic cues during text mining.
- Filter to high-confidence statements that have belief score > 0.95.
- Filter to the most specific version of statements in case a statement appears at multiple refinement levels.
- Filter strictly to genes in the Ras pathway (which are also the prior search terms around which The Ras Machine is built).
- Apply a set of semantic filters: filter phosphorylations to ones where the subject is a kinase, filter to amount regulation statements where the subject is a transcription factor, etc.
- Run the "Mechanism Linker" which applies logic over sets of statements to fill in missing information and remove certain redundancies as follows:
  - Find the most specific activity type known for each protein and "reduce" all active forms to that activity type. For example, if a protein is known to have generic "activity", but also "kinase" activity, and "kinase" activity is the only known specific activity type, then all the generic "activity" states will be reduced to "kinase" activity.
  - Find the most specific modifications known for each protein and "reduce" all modifications to that form.
  - Remove any activation statements that are redundant with respect to a modification and an active form statement. (For instance, if we know that A activates B, and also that A phosphorylates B, and Phosphorylated B is active, then we can remove the redundant A activates B statement.

- Rewrite all agents that appear in an active position in a statement ( e.g., A in the statement A activates B) to be in one of their known active forms. For example, if we have the statement A activates B, and we know that A is active when it's bound to C, then the statement is rewritten to A bound to C activates B.
- Filter out inconsequential modifications and activations, in other words, remove any statements that modify the state of an agent in a way that doesn't have any further downstream effect and is therefore inconsequential.

Having performed these steps, we were able to simulate the model using network-free stochastic simulation. Below is an example simulation trace for the amount of MAP2K1 phosphorylated on the S222 site:



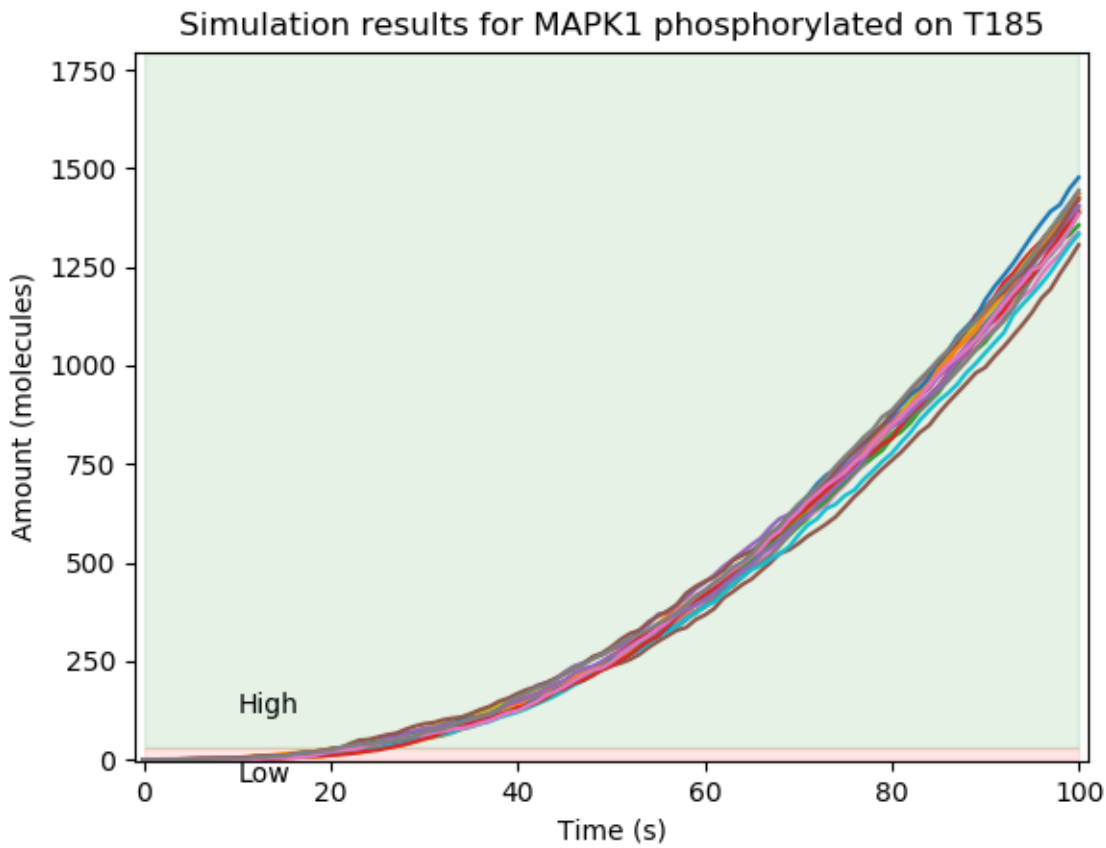

### Supporting network-free simulation

Until recently, EMMAA only supported deterministic ODE-based simulation for models. The main limitation is that ODE-based simulation requires fully defining the set of variables and equations representing system behavior up front. This implies that a reaction network (from which ODEs can be derived) needs to be generated, where the reaction network describes all biochemical reactions that change the amount or state of entities (typically proteins and small molecules). However, reaction networks can be very large (and potentially infinitely large) due to the combinatorial complexity of entities interacting with each other. (Consider for instance the trivial polymerization reaction where some entity X has two binding sites each of which can bind X, resulting in chains of X of unlimited length).

Network-free, agent-based simulation overcomes this challenge since it doesn't require enumerating all states up front, rather, one can provide an initial mixture from which the state of the system evolves as reaction events happen over time. To support this simulation mode, we integrated the Kappa simulator with EMMAA via the kappy Python

package. We implemented the API to the Kappa simulator such that it is consistent with the previous ODE-based simulator.

One specific example of a model which - due to combinatorial complexity - cannot be generated into a reaction network but can be simulated using this network-free approach is the Ras Model. The example below shows simulations of MAPK1 phosphorylated on T185.




### Adaptive sample-size dynamical property checking

One property of stochastic network-free simulation is that each simulation trace is different, and given any qualitative property, whether a trace satisfies that property or not can differ due to this stochasticity. So the question arises: how many simulations should one do to conclude - assuming pre-specified statistical error bounds - that a given property holds with at least a given probability. We integrated a sequential hypothesis testing algorithm with the property checking surrounding network-free simulation which can decide (after each simulation) whether to stop or to perform another simulation to determine the satisfaction of the property. This way, sample sizes are chosen adaptively and automatically in a principled way.

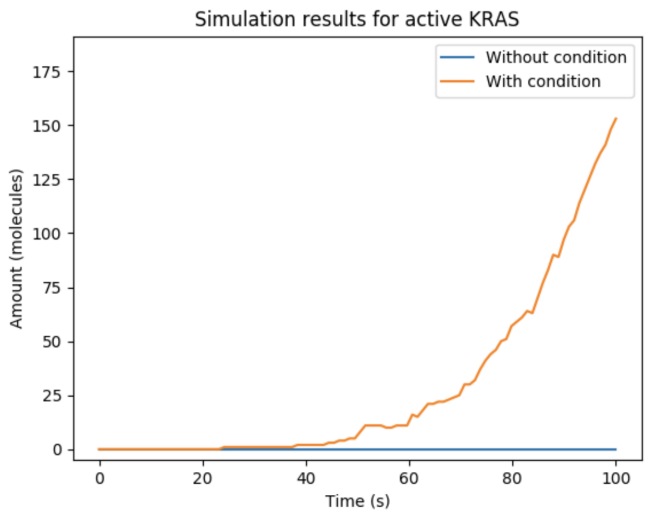
In the future, we will work on integrating parametric uncertainty in EMMAA model analysis. In that case, even deterministic ODE-based simulations will be subject to uncertainty, and the same sequential hypothesis testing approach will be applicable in that simulation mode too.

### Intervention-based dynamical queries

One of the query modes we have planned to support involves interventions where the amount of a given entity is modulated and the effect on a specific readout is examined. We implemented a query specification schema to describe such queries, implemented a new EMMAA Query UI tab for specifying these queries in and intuitive way, and integrated the back-end simulation engine supporting setting up, running, and then evaluating the results of such queries.

In the example below, we asked whether SOS1 leads to the activation of KRAS in a dynamical sense. This is evaluated by modulating the total amount of SOS1 between a high level (which can loosely model "overexpression") and a low level (which can loosely model "knock out"), and comparing the time course of active KRAS between the two conditions. In this case, we find that active KRAS is substantially higher when SOS1 is present at a high level, therefore the property is satisfied:

Query Results

| Query | Model | Result | Image |
|---|---|---|---|
| SOS1 increases active KRAS. | rasmodel | ✔ |  |

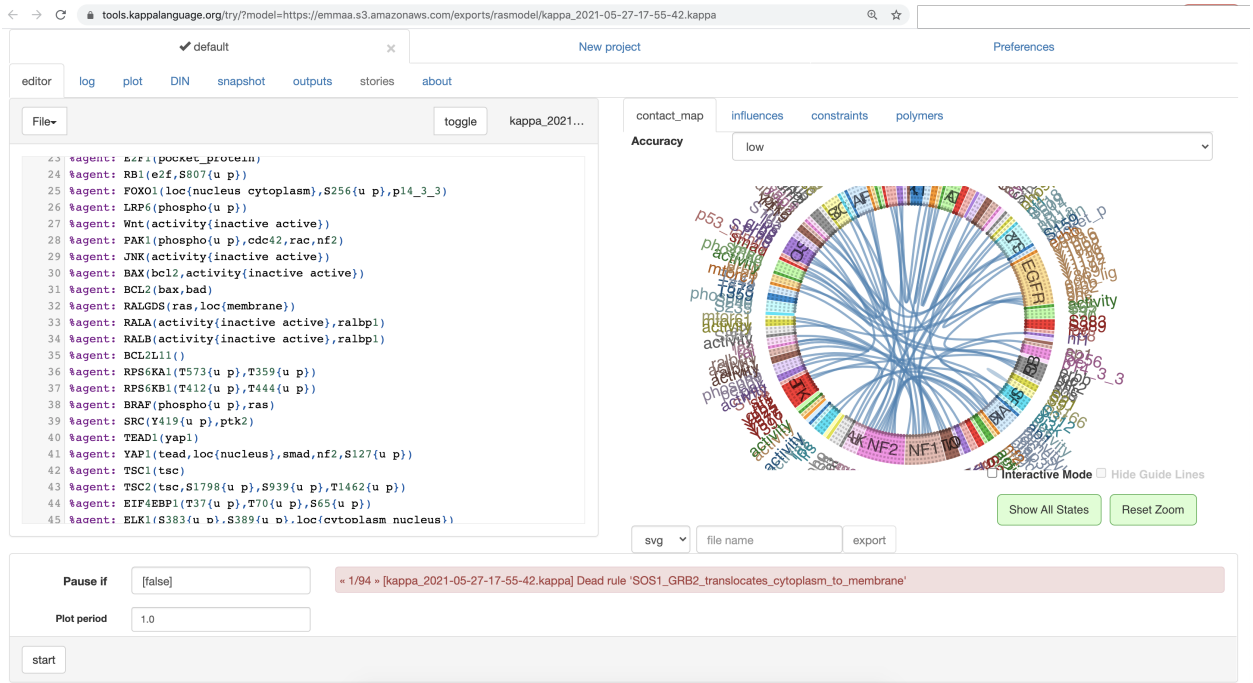


### Integration with the Kappa dynamical modeling and analysis UI

The team behind the Kappa language and tool set has developed a powerful integrated development environment for Kappa models using an easy-to-use web inteface which integrates panels for defining and modifying the model, and examining static analysis and simulation-based dynamical analysis results: https://tools.kappalanguage.org/try.

The Kappa UI supports loading models directly from URLs which allows straightforward integration with EMMAA. Namely, each EMMAA model (where this makes sense) is generated into a Kappa export after each daily model update, and these exports come with a stable URL. We now added a link out to the Kappa UI for each model where such an export is available, allowing users to perform analysis on that interface.

The screenshot below shows the Ras Model in the Kappa UI. On the left, the Kappa export of the model can be edited directly. On the right, the contact map (one of the static analysis outputs) is shown, and in the bottom, warning messages about "dead rules" (rules that are inconsequential from a dynamical perspective) can be browsed. Numerous further tabs support a variety of other analysis modes.




## 6.8.2 Improved EMMAA query UI and REST API

This month we made various changes to the EMMAA service to improve user experience for both users querying models via the interactive web interface and through the REST API programmatically.

Having added intervention-based dynamical queries as described in the previous section, we now support four types of queries. We updated the names of the query types to be more descriptive and added instructions on how to submit and interpret the results for each of the query types using the EMMAA interactive query tool.

Over the last several months we reported adding various endpoints to the EMMAA REST API to facilitate integration with the Uncharted UI. During this reporting period we extended and improved the REST API and added an automatically rendered Swagger documentation that describes the methods, input requirements, and expected responses for each endpoint. We grouped the endpoints into three categories corresponding to the goals for which they can be used (retrieving EMMAA models' metadata, getting latest models' updates, and running EMMAA queries). The previous */run_query* endpoint that allowed running any type of query was replaced with four separate endpoints for each of the query types for convinience and better validation of user input.

The documentation contains the descriptions and example values for each parameter that a given endpoint can accept. The interactive Swagger documentation also allows manually modifying the example input and trying out the endpoints.

In addition, we provide examples and descriptions for responses to validate the output and facilitate the interpretation of results.

## 6.8.3 Network representation learning for EMMAA models

Sets of INDRA statements such as those associated with each EMMAA model can be assembled into graph-like data structures of decreasing granularity: directed graphs with typed edges, directed graphs without typed edges, and ultimately, undirected graphs. Different network representation learning methods can be used for each data structure to assign dense vectors to nodes (and edges, if applicable). These are useful for downstream machine learning tasks (e.g., clustering, classification, regression), link prediction, and entity disambiguation. Our goal is to use the representations to investigate the similarities between nodes' representations between the full INDRA database and each EMMAA

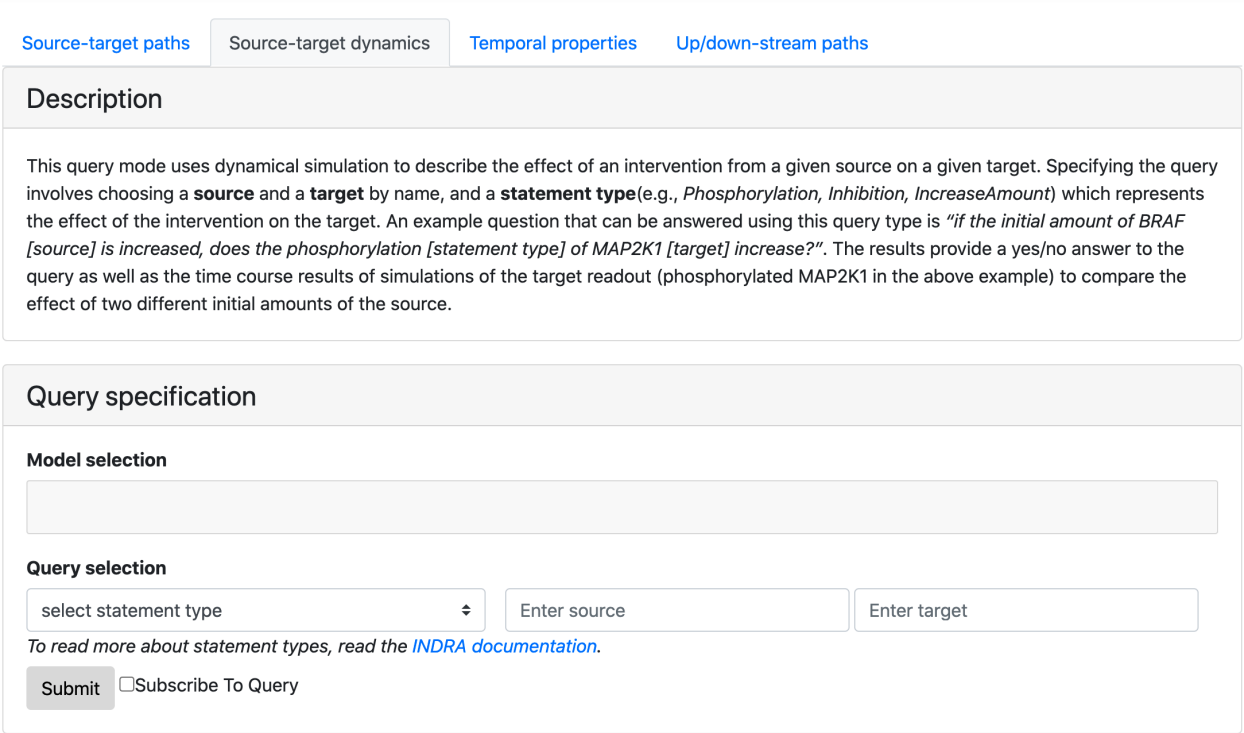



Fig. 6: *Query page showing four types of queries, description and the form*

model to identify context-specific nodes as well as to make recommendations for including or removing nodes from each EMMAA model.

### Building a preliminary NRL pipeline

There are both practical and theoretical considerations for using the highest granular directed graphs with typed edges (i.e., knowledge graphs). Most of the associated methods, called knowledge graph embedding models (KGEMs), suffer from issues in scalability. Because most useful biological networks are larger than the size supported, there is still minimal theoretical insight into how the methods perform on biological networks, which have very different topology to the *semantic web* datasets to which they are typically applied and evaluated.

Instead, we built a reproducible pipeline for assembling the full INDRA database and each EMMAA model into directed graphs without typed edges at varying belief levels for application of the *node2vec* random walk embedding model to generate 64-dimensional vectors in Euclidean space for each node.

Later, we will automate this pipeline to run automatically upon each update to the full INDRA Database and each EMMAA model such that the latest information can be incorporated. Further, the results could be included in EMMAA API endpoint that returns model-specific metadata for each node.

### Comparing EMMAA models with background knowledge

We first investigated where nodes from each EMMAA model appear in the embedding space generated from the full INDRA database with a belief greater than 60%. We used principal component analysis to project into 2-dimensional space for visualization. Because of the formulation of the *node2vec* method, each features' contributions to the overall variance are more homogenous than typical feature sets. The first two principle components only explained ~35%

# EMMAA REST API 1.0

[ Base URL: / ]
/swagger.json

EMMAA REST API

## Metadata Get EMMAA models metadata

| | | |
|---|---|---|
| GET | /metadata/entity_info/{model} | Get information about an entity |
| GET | /metadata/model_info/{model} | Get metadata for model |
| GET | /metadata/models | Get a list of all available models |
| GET | /metadata/test_corpora/{model} | Get a list of available test corpora for model |
| GET | /metadata/tests_info/{test_corpus} | Get test corpus metadata |

## Latest Get updates specific to latest models

| | | |
|---|---|---|
| GET | /latest/curated_statements/{model} | Get hashes of curated statements by category |
| GET | /latest/statements/{model} | Return model latest statements and link to S3 latest statements file |
| GET | /latest/statements_url/{model} | Return a link to model latest statements file on S3 |
| GET | /latest/stats_date | Get latest date for which both model and test stats are available |

## Query Run EMMAA queries

| | | |
|---|---|---|
| POST | /query/source_target_dynamic | Simulate a model to describe the effect of an intervention |
| POST | /query/source_target_path | Explain an effect between source and target |
| POST | /query/temporal_dynamic | Simulate a model to verify if a certain pattern is met |
| POST | /query/up_down_stream_path | Find causal paths to or from a given entity |

Fig. 7: *EMMAA REST API endpoints*

POST /query/up_down_stream_path Find causal paths to or from a given entity

Parameters

Try it out

Name | Description

payload * required
object
(body)

Example Value | Model

```
{
  "model": "rasmodel",
  "entity": {
    "type": "Agent",
    "name": "AKT1",
    "db_refs": {
      "HGNC": "391"
    }
  },
  "entity_role": "object",
  "stmt_type": "Activation",
  "terminal_ns": [
    "HGNC",
    "UP",
    "FPLX"
  ]
}
```

Parameter content type

application/json

payload * required
object
(body)

Example Value | Model

```
open_query { 
  model                 string
                        example: rasmodel

                        A name of EMMAA model to query (e.g. aml, covid19)

  entity                { 
                          description:
                                        INDRA Agent JSON to start the search from.

                        }
                        example: OrderedMap { "type": "Agent", "name": "AKT1", "db_refs": OrderedMap { "HGNC": "391" } }
  entity_role           string
                        example: object

                        subject for downstream or object for upstream search.

  stmt_type             string
                        example: Activation

                        Type of effect to search for.

  terminal_ns           [ 
                        example: List [ "HGNC", "UP", "FPLX" ]

                        Optional list of namespaces to constrain the types of up/downstream entities

                        string]
}
```

Fig. 8: *Example input and parameters description for Up/down-stream query endpoint*

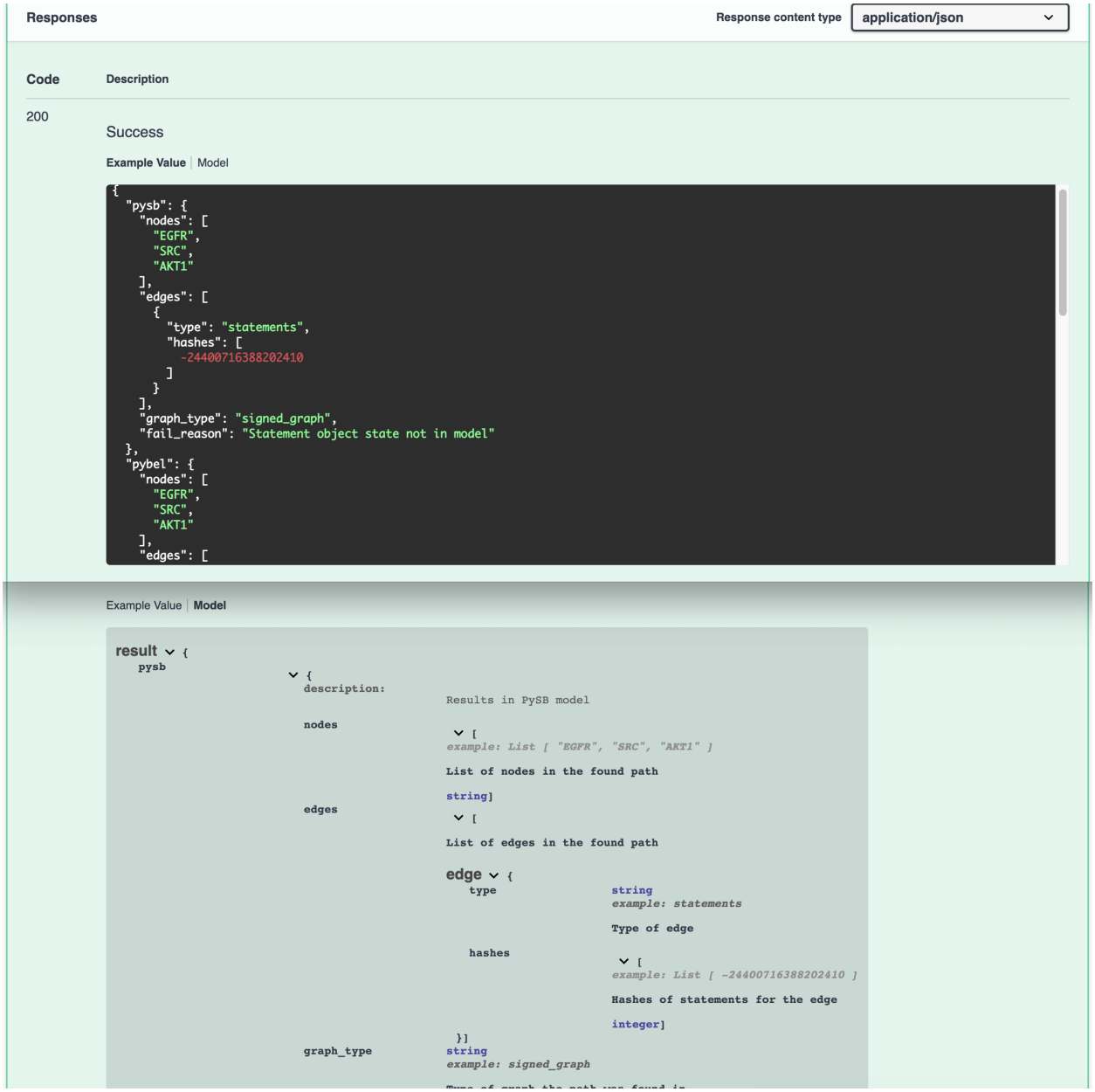
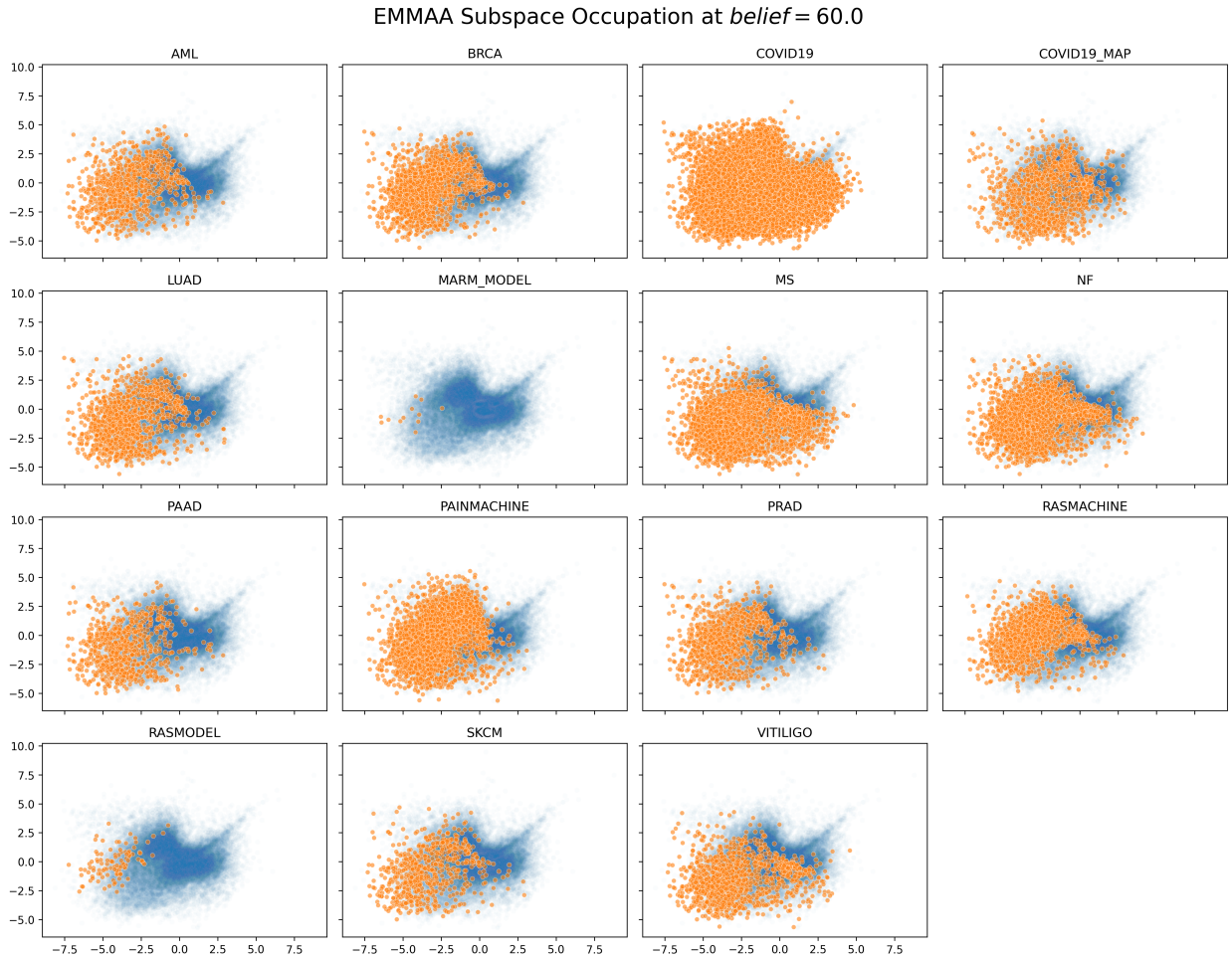



Fig. 9: *Example response and interpretation*

of the variance. Background nodes are shown with low opacity in blue while EMMAA nodes are shown with high opacity in orange.

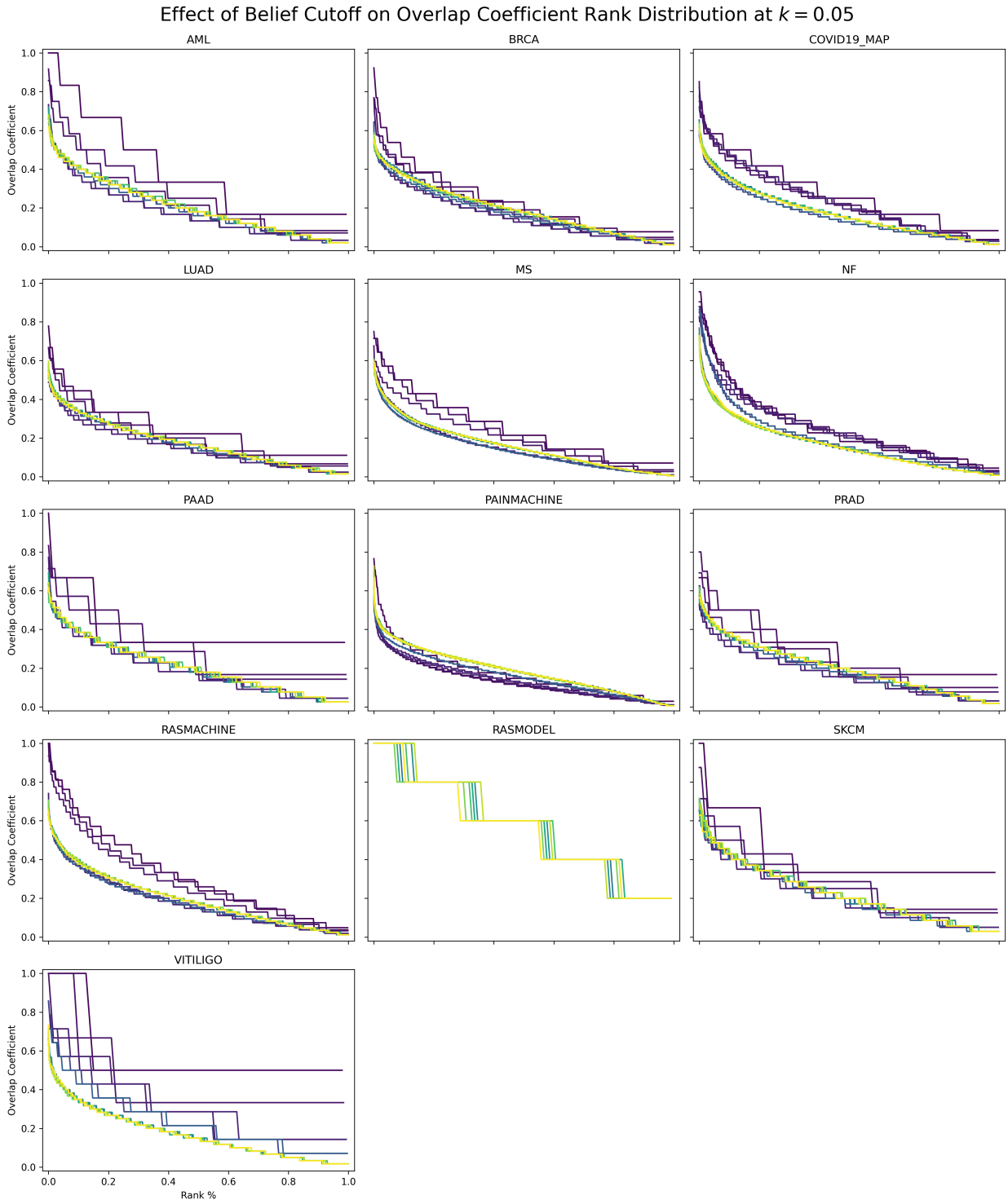



Interestingly, there are some regions that are not covered by any EMMAA model. While this could be because of a bias in the contexts covered by current EMMAA models, it might also lead to insight in underrepresented biology.

### Identification of context-specific nodes

Next, we wanted to identify nodes with the most similar and most dissimilar topologies in the INDRA database and a given EMMAA model. We hypothesize that the most similar nodes represent the most generic biology and the most dissimilar nodes represent context-specific biology. We investigated the overlap between the k-nearest neighbors in embedding space for each node in the INDRA Database with the k-nearest neighbors in the embedding space for each EMMAA model. To account for the size differences in the INDRA database and much smaller EMMAA models, we used a fractional k=0.05 and the set overlap coefficient, which is more appropriate for sets of different sizes. We performed the same task on the embeddings generated based on several belief cutoffs.

The following chart shows that when the belief cutoff is increased, the shape of the overlap coefficient rank distribution typically shifts towards higher overlap coefficients. Darker lines correspond to higher belief. Notably, this pattern does not hold for the literature derived models (e.g., Pain Model). The RAS Model results should also be disregarded since the statements there should have an axiomatic belief of 1.0, but are tagged via TRIPS so have a lower belief.




The nodes in the long tail of these distributions hold the most potential for novelty but also the most liability for irrelevance. Our next step is to build a minimal browser for looking into these nodes as having a human in the loop for the investigation of these nodes at the boundaries of EMMAA models could be useful.

### Towards an automated recommendation engine

Our ongoing work towards an automated recommendation looks at the neighbors of nodes in the EMMAA models within the embedding space from the full INDRA Database to identify potential additions. We are investigate several clustering algorithms and their classification counterparts as potential methods for scoring nodes for inclusion. Similarly, we are investigating anomaly detection methods at can be used in reverse towards the same goal.

Later, we will return to the k-nearest neighbors analysis to identify nodes that could potentially be removed from a given EMMAA model.

### Improvements to `pykeen`

While *node2vec* performs well on biological networks due to the symmetry in the model formulation and the important property of local community structure common to biological networks, we would still like to use more powerful methods for network representation learning. We are making improvements to the `pykeen` package for knowledge graph embeddings in order to make it more scalable and applicable for the directed graph with typed edges assembly of INDRA statements. So far, we have made several improvements to its memory management on large graphs and begun work integrating the `accelerate` for scaling across multiple GPUs.

- genindex
- modindex
- search

# Python Module Index

## e


emmaa.analyze_tests_results, 49
emmaa.answer_queries, 55
emmaa.aws_lambda_functions, 64
emmaa.aws_lambda_functions.after_update, 65
emmaa.aws_lambda_functions.model_manager_update, 65
emmaa.aws_lambda_functions.model_queries, 68
emmaa.aws_lambda_functions.model_stats, 67
emmaa.aws_lambda_functions.model_tests, 66
emmaa.aws_lambda_functions.model_updates, 64
emmaa.aws_lambda_functions.test_pipeline, 66
emmaa.aws_lambda_functions.test_stats, 67
emmaa.aws_lambda_functions.test_update, 69
emmaa.aws_lambda_functions.test_update_pipeline, 68
emmaa.aws_lambda_functions.update_lambda, 69
emmaa.aws_lambda_functions.update_pipeline, 64
emmaa.db, 60
emmaa.db.manager, 62
emmaa.db.schema, 60
emmaa.filter_functions, 77
emmaa.model, 40
emmaa.model_tests, 45
emmaa.priors, 55
emmaa.priors.cancer_prior, 57
emmaa.priors.gene_list_prior, 57
emmaa.priors.literature_prior, 56
emmaa.priors.prior_stmts, 59
emmaa.priors.reactome_prior, 58
emmaa.queries, 53
emmaa.readers, 59
emmaa.readers.aws_reader, 59
emmaa.readers.db_client_reader, 60
emmaa.statements, 39
emmaa.subscription, 70
emmaa.subscription.email_service, 73
emmaa.subscription.email_util, 75
emmaa.subscription.notifications, 70
emmaa.util, 76
emmaa.xdd, 69
emmaa.xdd.xdd_client, 69

# Index


## A

add_emmaa_annotations() (*in module emmaa.statements*), 39
add_paper_ids() (*emmaa.model.EmmaaModel method*), 41
add_result() (*emmaa.model_tests.ModelManager method*), 46
add_statements() (*emmaa.model.EmmaaModel method*), 41
add_test() (*emmaa.model_tests.ModelManager method*), 46
add_user() (*emmaa.db.manager.EmmaaDatabaseManager method*), 62
agent_from_gene_name() (*in module emmaa.priors.gene_list_prior*), 58
answer_dynamic_query() (*emmaa.model_tests.ModelManager method*), 46
answer_immediate_query() (*emmaa.answer_queries.QueryManager method*), 55
answer_intervention_query() (*emmaa.model_tests.ModelManager method*), 46
answer_open_query() (*emmaa.model_tests.ModelManager method*), 46
answer_path_query() (*emmaa.model_tests.ModelManager method*), 46
answer_queries() (*emmaa.model_tests.ModelManager method*), 46
answer_queries_from_s3() (*in module emmaa.answer_queries*), 55
answer_registered_queries() (*emmaa.answer_queries.QueryManager method*), 55
applicable() (*emmaa.model_tests.RefinementTestConnector static method*), 47
applicable() (*emmaa.model_tests.ScopeTestConnector static method*), 47
applicable() (*emmaa.model_tests.TestConnector static method*), 48
applicable_tests (*emmaa.model_tests.ModelManager attribute*), 46
assemble_dynamic_pysb() (*emmaa.model.EmmaaModel method*), 41
assemble_pybel() (*emmaa.model.EmmaaModel method*), 41
assemble_pysb() (*emmaa.model.EmmaaModel method*), 41
assemble_signed_graph() (*emmaa.model.EmmaaModel method*), 42
assemble_unsigned_graph() (*emmaa.model.EmmaaModel method*), 42
assembled_stmts (*emmaa.model.EmmaaModel attribute*), 41
assembly_config (*emmaa.model.EmmaaModel attribute*), 41

## C

check() (*emmaa.model_tests.StatementCheckingTest method*), 47
check_stmt() (*in module emmaa.statements*), 39
close_to_quota_max() (*in module emmaa.subscription.email_service*), 73
ComparativeInterventionProperty (*class in emmaa.queries*), 53
create_tables() (*emmaa.db.manager.EmmaaDatabaseManager method*), 62

## D

date_str (*emmaa.model_tests.ModelManager attribute*), 46
dbes_exist() (*in module emmaa.util*), 76

drop_tables() (*emmaa.db.manager.EmmaaDatabaseManager method*), 62
DynamicProperty (*class in emmaa.queries*), 53

## E

eliminate_copies() (*emmaa.model.EmmaaModel method*), 42
EmailHtmlBody (*class in emmaa.subscription.notifications*), 70
emmaa.analyze_tests_results (*module*), 49
emmaa.answer_queries (*module*), 55
emmaa.aws_lambda_functions (*module*), 64
emmaa.aws_lambda_functions.after_update (*module*), 65
emmaa.aws_lambda_functions.model_manager_update (*module*), 65
emmaa.aws_lambda_functions.model_queries (*module*), 68
emmaa.aws_lambda_functions.model_stats (*module*), 67
emmaa.aws_lambda_functions.model_tests (*module*), 66
emmaa.aws_lambda_functions.model_updates (*module*), 64
emmaa.aws_lambda_functions.test_pipeline (*module*), 66
emmaa.aws_lambda_functions.test_stats (*module*), 67
emmaa.aws_lambda_functions.test_update (*module*), 69
emmaa.aws_lambda_functions.test_update_pipeline (*module*), 68
emmaa.aws_lambda_functions.update_lambda (*module*), 69
emmaa.aws_lambda_functions.update_pipeline (*module*), 64
emmaa.db (*module*), 60
emmaa.db.manager (*module*), 62
emmaa.db.schema (*module*), 60
emmaa.filter_functions (*module*), 77
emmaa.model (*module*), 40
emmaa.model_tests (*module*), 45
emmaa.priors (*module*), 55
emmaa.priors.cancer_prior (*module*), 57
emmaa.priors.gene_list_prior (*module*), 57
emmaa.priors.literature_prior (*module*), 56
emmaa.priors.prior_stmts (*module*), 59
emmaa.priors.reactome_prior (*module*), 58
emmaa.queries (*module*), 53
emmaa.readers (*module*), 59
emmaa.readers.aws_reader (*module*), 59
emmaa.readers.db_client_reader (*module*), 60
emmaa.statements (*module*), 39
emmaa.subscription (*module*), 70
emmaa.subscription.email_service (*module*), 73
emmaa.subscription.email_util (*module*), 75
emmaa.subscription.notifications (*module*), 70
emmaa.util (*module*), 76
emmaa.xdd (*module*), 69
emmaa.xdd.xdd_client (*module*), 69
EmmaaDatabaseError, 64
EmmaaDatabaseManager (*class in emmaa.db.manager*), 62
EmmaaModel (*class in emmaa.model*), 40
EmmaaStatement (*class in emmaa.statements*), 39
EmmaaTest (*class in emmaa.model_tests*), 45
english_test_results (*emmaa.analyze_tests_results.TestRound attribute*), 52
entities (*emmaa.model_tests.ModelManager attribute*), 46
extend_unique() (*emmaa.model.EmmaaModel method*), 42

## F

filter_chem_mesh_go() (*in module emmaa.filter_functions*), 77
filter_emmaa_stmts_by_metadata() (*in module emmaa.statements*), 40
filter_indra_stmts_by_metadata() (*in module emmaa.statements*), 40
filter_to_internal_edges() (*in module emmaa.filter_functions*), 77
find_delta_hashes() (*emmaa.analyze_tests_results.Round method*), 51
find_drugs_for_genes() (*emmaa.priors.cancer_prior.TcgaCancerPrior static method*), 57
find_drugs_for_genes() (*in module emmaa.priors.reactome_prior*), 58
find_latest_emails() (*in module emmaa.util*), 76
find_latest_s3_file() (*in module emmaa.util*), 76
find_latest_s3_files() (*in module emmaa.util*), 76
find_nth_latest_s3_file() (*in module emmaa.util*), 76
format_results() (*in module emmaa.answer_queries*), 55
from_json() (*emmaa.priors.SearchTerm class method*), 56
function_mapping (*emmaa.analyze_tests_results.Round attribute*),

50

## G

GeneListPrior (*class in emmaa.priors.gene_list_prior*), 57
generate_signature() (*in module emmaa.subscription.email_util*), 75
generate_stats_on_s3() (*in module emmaa.analyze_tests_results*), 53
generate_unsubscribe_link() (*in module emmaa.subscription.email_util*), 75
generate_unsubscribe_qs() (*in module emmaa.subscription.email_util*), 75
get_agent_distribution() (*emmaa.analyze_tests_results.ModelRound method*), 49
get_agent_from_gilda() (*in module emmaa.queries*), 54
get_agent_from_text() (*in module emmaa.queries*), 54
get_agent_from_trips() (*in module emmaa.queries*), 54
get_all_raw_paper_ids() (*emmaa.analyze_tests_results.ModelRound method*), 49
get_all_result_hashes() (*emmaa.db.manager.EmmaaDatabaseManager method*), 62
get_all_update_messages() (*in module emmaa.subscription.notifications*), 71
get_applied_test_hashes() (*emmaa.analyze_tests_results.TestRound method*), 52
get_assembled_entities() (*emmaa.model.EmmaaModel method*), 42
get_assembled_statements() (*in module emmaa.model*), 44
get_assembled_stmts_by_paper() (*emmaa.analyze_tests_results.ModelRound method*), 49
get_document_figures() (*in module emmaa.xdd.xdd_client*), 69
get_document_objects() (*in module emmaa.xdd.xdd_client*), 70
get_drugs_for_gene() (*in module emmaa.priors*), 56
get_email_subscriptions() (*in module emmaa.subscription.email_util*), 76
get_english_statements_by_hash() (*emmaa.analyze_tests_results.ModelRound method*), 49
get_entities() (*emmaa.model.EmmaaModel method*), 42
get_entities() (*emmaa.model_tests.EmmaaTest method*), 45
get_entities() (*emmaa.model_tests.StatementCheckingTest method*), 47
get_entities() (*emmaa.queries.PathProperty method*), 54
get_figures_from_objects() (*in module emmaa.xdd.xdd_client*), 70
get_figures_from_query() (*in module emmaa.xdd.xdd_client*), 70
get_genes_contained_in_pathway (*in module emmaa.priors.reactome_prior*), 58
get_indra_stmts() (*emmaa.model.EmmaaModel method*), 42
get_model_deltas() (*in module emmaa.subscription.notifications*), 71
get_model_stats() (*in module emmaa.model*), 44
get_model_users() (*emmaa.db.manager.EmmaaDatabaseManager method*), 62
get_mutated_genes() (*emmaa.priors.cancer_prior.TcgaCancerPrior method*), 57
get_new_readings() (*emmaa.model.EmmaaModel method*), 42
get_number_passed_tests() (*emmaa.analyze_tests_results.TestRound method*), 52
get_number_raw_papers() (*emmaa.analyze_tests_results.ModelRound method*), 49
get_paper_ids_from_stmts() (*emmaa.model.EmmaaModel method*), 42
get_paper_titles_and_links() (*emmaa.analyze_tests_results.ModelRound method*), 49
get_papers_distribution() (*emmaa.analyze_tests_results.ModelRound method*), 49
get_passed_test_hashes() (*emmaa.analyze_tests_results.TestRound method*), 52
get_pathways_containing_gene (*in module emmaa.priors.reactome_prior*), 58
get_queries() (*emmaa.db.manager.EmmaaDatabaseManager method*), 62
get_raw_statements_for_pmids() (*in module emmaa.priors.literature_prior*), 56
get_registered_queries() (*emmaa.answer_queries.QueryManager method*), 55
get_relevant_nodes() (*em-*

*maa.priors.cancer_prior.TcgaCancerPrior method*), 57
get_results() (*emmaa.db.manager.EmmaaDatabaseManager method*), 62
get_s3_client() (*in module emmaa.util*), 76
get_send_statistics() (*in module emmaa.subscription.email_service*), 73
get_statement_types() (*emmaa.analyze_tests_results.ModelRound method*), 49
get_statements_by_evidence() (*emmaa.analyze_tests_results.ModelRound method*), 49
get_stmt_hashes() (*emmaa.analyze_tests_results.ModelRound method*), 49
get_stmts_for_gene() (*in module emmaa.priors.prior_stmts*), 59
get_stmts_for_gene_list() (*in module emmaa.priors.prior_stmts*), 59
get_subscribed_queries() (*emmaa.db.manager.EmmaaDatabaseManager method*), 62
get_subscribed_users() (*emmaa.db.manager.EmmaaDatabaseManager method*), 63
get_temporal_pattern() (*emmaa.queries.DynamicProperty method*), 53
get_total_applied_tests() (*emmaa.analyze_tests_results.TestRound method*), 52
get_total_statements() (*emmaa.analyze_tests_results.ModelRound method*), 49
get_updated_mc() (*emmaa.model_tests.ModelManager method*), 47
get_user_models() (*emmaa.db.manager.EmmaaDatabaseManager method*), 63
get_user_query_delta() (*in module emmaa.subscription.notifications*), 71
GroundingError, 53

## H

hash_response_list() (*emmaa.model_tests.ModelManager method*), 47

## I

is_internal() (*in module emmaa.statements*), 40

## J

json_stats (*emmaa.analyze_tests_results.ModelStatsGenerator attribute*), 50
json_stats (*emmaa.analyze_tests_results.StatsGenerator attribute*), 51
json_stats (*emmaa.analyze_tests_results.TestStatsGenerator attribute*), 52

## L

lambda_handler() (*in module emmaa.aws_lambda_functions.after_update*), 65
lambda_handler() (*in module emmaa.aws_lambda_functions.model_manager_update*), 65
lambda_handler() (*in module emmaa.aws_lambda_functions.model_queries*), 68
lambda_handler() (*in module emmaa.aws_lambda_functions.model_stats*), 67
lambda_handler() (*in module emmaa.aws_lambda_functions.model_tests*), 66
lambda_handler() (*in module emmaa.aws_lambda_functions.model_updates*), 64
lambda_handler() (*in module emmaa.aws_lambda_functions.test_pipeline*), 66
lambda_handler() (*in module emmaa.aws_lambda_functions.test_stats*), 67
lambda_handler() (*in module emmaa.aws_lambda_functions.test_update*), 69
lambda_handler() (*in module emmaa.aws_lambda_functions.test_update_pipeline*), 68
lambda_handler() (*in module emmaa.aws_lambda_functions.update_pipeline*), 64
last_updated_date() (*in module emmaa.model*), 45
load_config_from_s3() (*in module emmaa.model*), 45
load_from_s3() (*emmaa.model.EmmaaModel class method*), 42
load_sif_prior() (*emmaa.priors.cancer_prior.TcgaCancerPrior method*), 57
load_stmts_from_s3() (*in module emmaa.model*), 45
load_tests_from_s3() (*in module emmaa.model_tests*), 48

## M


make_changes_over_time() (*emmaa.analyze_tests_results.ModelStatsGenerator method*), 50
make_changes_over_time() (*emmaa.analyze_tests_results.StatsGenerator method*), 51
make_changes_over_time() (*emmaa.analyze_tests_results.TestStatsGenerator method*), 52
make_config() (*emmaa.priors.gene_list_prior.GeneListPrior method*), 58
make_curation_summary() (*emmaa.analyze_tests_results.ModelStatsGenerator method*), 50
make_date_str() (*in module emmaa.util*), 77
make_gene_statements() (*emmaa.priors.gene_list_prior.GeneListPrior method*), 58
make_html_report_per_user() (*in module emmaa.subscription.notifications*), 72
make_model() (*emmaa.priors.gene_list_prior.GeneListPrior method*), 58
make_model_delta() (*emmaa.analyze_tests_results.ModelStatsGenerator method*), 50
make_model_html_email() (*in module emmaa.subscription.notifications*), 72
make_model_summary() (*emmaa.analyze_tests_results.ModelStatsGenerator method*), 50
make_paper_delta() (*emmaa.analyze_tests_results.ModelStatsGenerator method*), 50
make_paper_summary() (*emmaa.analyze_tests_results.ModelStatsGenerator method*), 50
make_prior() (*emmaa.priors.cancer_prior.TcgaCancerPrior method*), 57
make_prior_from_genes() (*in module emmaa.priors.reactome_prior*), 58
make_reports_from_results() (*in module emmaa.subscription.notifications*), 72
make_search_terms() (*emmaa.priors.gene_list_prior.GeneListPrior method*), 58
make_search_terms() (*in module emmaa.priors.literature_prior*), 56
make_stats() (*emmaa.analyze_tests_results.ModelStatsGenerator method*), 50
make_stats() (*emmaa.analyze_tests_results.TestStatsGenerator method*), 53
make_str_report_per_user() (*in module emmaa.subscription.notifications*), 72
make_test_summary() (*emmaa.analyze_tests_results.TestStatsGenerator method*), 53
make_tests() (*emmaa.model_tests.TestManager method*), 48
make_tests_delta() (*emmaa.analyze_tests_results.TestStatsGenerator method*), 53
mc_mapping (*emmaa.model_tests.ModelManager attribute*), 46
mc_types (*emmaa.model_tests.ModelManager attribute*), 46
mc_types_results (*emmaa.analyze_tests_results.TestRound attribute*), 52
model_to_tests() (*in module emmaa.model_tests*), 48
model_update_notify() (*in module emmaa.subscription.notifications*), 73
ModelDeltaEmailHtmlBody (*class in emmaa.subscription.notifications*), 70
ModelManager (*class in emmaa.model_tests*), 46
ModelRound (*class in emmaa.analyze_tests_results*), 49
ModelStatsGenerator (*class in emmaa.analyze_tests_results*), 49


## N


ndex_network (*emmaa.model.EmmaaModel attribute*), 41
NotAClassName, 76


## O


OpenSearchQuery (*class in emmaa.queries*), 53


## P


passed_over_total() (*emmaa.analyze_tests_results.TestRound method*), 52
path_stmt (*emmaa.queries.OpenSearchQuery attribute*), 54
path_stmt_types (*emmaa.model_tests.ModelManager attribute*), 46
PathProperty (*class in emmaa.queries*), 54
process_response() (*emmaa.model_tests.ModelManager method*), 47
put_queries() (*emmaa.db.manager.EmmaaDatabaseManager method*), 63

put_results() (*emmaa.db.manager.EmmaaDatabaseManager method*), 63

## Q

Query (*class in emmaa.db.schema*), 60
Query (*class in emmaa.queries*), 54
query_config (*emmaa.model.EmmaaModel attribute*), 41
QueryEmailHtmlBody (*class in emmaa.subscription.notifications*), 71
QueryManager (*class in emmaa.answer_queries*), 55

## R

read_db_doi_search_terms() (*in module emmaa.readers.db_client_reader*), 60
read_db_ids_search_terms() (*in module emmaa.readers.db_client_reader*), 60
read_db_pmid_search_terms() (*in module emmaa.readers.db_client_reader*), 60
read_pmid_search_terms() (*in module emmaa.readers.aws_reader*), 59
read_pmids() (*in module emmaa.readers.aws_reader*), 59
reading_config (*emmaa.model.EmmaaModel attribute*), 41
RefinementTestConnector (*class in emmaa.model_tests*), 47
register_email_unsubscribe() (*in module emmaa.subscription.email_util*), 76
register_filter() (*in module emmaa.filter_functions*), 78
render() (*emmaa.subscription.notifications.ModelDeltaEmailHtmlBody method*), 70
render() (*emmaa.subscription.notifications.QueryEmailHtmlBody method*), 71
Result (*class in emmaa.db.schema*), 61
results_to_json() (*emmaa.model_tests.ModelManager method*), 47
retrieve_results_from_hashes() (*emmaa.answer_queries.QueryManager method*), 55
Round (*class in emmaa.analyze_tests_results*), 50
run_all_tests() (*emmaa.model_tests.ModelManager method*), 47
run_assembly() (*emmaa.model.EmmaaModel method*), 42
run_model_tests_from_s3() (*in module emmaa.model_tests*), 48
run_tests() (*emmaa.model_tests.TestManager method*), 48
run_tests_per_mc() (*emmaa.model_tests.ModelManager method*), 47
rx_id_from_up_id (*in module emmaa.priors.reactome_prior*), 58

## S

save_assembled_statements() (*emmaa.model_tests.ModelManager method*), 47
save_config_to_s3() (*in module emmaa.model*), 45
save_tests_to_s3() (*in module emmaa.model_tests*), 48
save_to_s3() (*emmaa.model.EmmaaModel method*), 42
ScopeTestConnector (*class in emmaa.model_tests*), 47
search_biorxiv() (*emmaa.model.EmmaaModel static method*), 42
search_elsevier() (*emmaa.model.EmmaaModel static method*), 43
search_literature() (*emmaa.model.EmmaaModel method*), 43
search_pubmed() (*emmaa.model.EmmaaModel static method*), 43
search_terms (*emmaa.model.EmmaaModel attribute*), 41
search_terms_from_nodes() (*emmaa.priors.cancer_prior.TcgaCancerPrior static method*), 57
SearchTerm (*class in emmaa.priors*), 55
send_document_search_request() (*in module emmaa.xdd.xdd_client*), 70
send_email() (*in module emmaa.subscription.email_service*), 74
send_query_search_request() (*in module emmaa.xdd.xdd_client*), 70
send_request() (*in module emmaa.xdd.xdd_client*), 70
SimpleInterventionProperty (*class in emmaa.queries*), 54
sort_s3_files_by_date_str() (*in module emmaa.util*), 77
sort_s3_files_by_last_mod() (*in module emmaa.util*), 77
StatementCheckingTest (*class in emmaa.model_tests*), 47
StatsGenerator (*class in emmaa.analyze_tests_results*), 51
stmts (*emmaa.model.EmmaaModel attribute*), 41
stmts_by_papers (*emmaa.analyze_tests_results.ModelRound attribute*), 49

strip_out_date() (*in module emmaa.util*), 77
StructuralProperty (*class in emmaa.queries*), 54
subscribe_to_model() (*emmaa.db.manager.EmmaaDatabaseManager method*), 63

## T

TcgaCancerPrior (*class in emmaa.priors.cancer_prior*), 57
test_config (*emmaa.model.EmmaaModel attribute*), 41
TestConnector (*class in emmaa.model_tests*), 47
TestManager (*class in emmaa.model_tests*), 48
TestRound (*class in emmaa.analyze_tests_results*), 51
tests (*emmaa.analyze_tests_results.TestRound attribute*), 52
TestStatsGenerator (*class in emmaa.analyze_tests_results*), 52
to_emmaa_stmts() (*in module emmaa.statements*), 40
to_json() (*emmaa.model.EmmaaModel method*), 43
to_json() (*emmaa.priors.SearchTerm method*), 56
tweet_deltas() (*in module emmaa.subscription.notifications*), 73

## U

up_id_from_rx_id (*in module emmaa.priors.reactome_prior*), 59
update_email_subscription() (*emmaa.db.manager.EmmaaDatabaseManager method*), 63
update_from_disease_map() (*emmaa.model.EmmaaModel method*), 43
update_from_files() (*emmaa.model.EmmaaModel method*), 43
update_to_ndex() (*emmaa.model.EmmaaModel method*), 44
update_with_cord19() (*emmaa.model.EmmaaModel method*), 44
upload_function() (*in module emmaa.aws_lambda_functions.update_lambda*), 69
upload_results() (*emmaa.model_tests.ModelManager method*), 47
upload_to_ndex() (*emmaa.model.EmmaaModel method*), 44
User (*class in emmaa.db.schema*), 60
UserModel (*class in emmaa.db.schema*), 61
UserQuery (*class in emmaa.db.schema*), 60

## V

verify_email_signature() (*in module emmaa.subscription.email_util*), 76