Adapting REACH for Virus Information

Matthew Peterson, Tonia Korves. The MITRE Corporation

This document describes the process we used for adapting the REACH code to include entities, events, and a reading pipeline to support extraction of information about vaccines and viruses in support of a MITRE research project.

Structuring the Project

For rapid prototyping, we forked the REACH code directly. As the project continued, we tried to separate as much of our code as possible into separate classes and packages rather than editing existing code. The goal was to utilize REACH functionality in a separate application, rather than to modify REACH. There were technical hurdles that prevented us from achieving this goal. These are described in a later section of this document. Instead, we made modifications to the REACH application to perform the extraction tasks for our project.

This included the following:

- Put custom rules into a separate package, and referenced existing REACH rules with import statements
- Created a copy of the ReachCLI class to add specific command-line functionality needed for the workflow
- Wrote a custom RuleReader class to load our custom rules and a subset of the existing REACH rules
- Created a copy of the ReachSystem class, removing some more advanced functionality not needed for our reading workflow

Adding Support for New Entities

Updating Taxonomy

The REACH taxonomy focuses on a number of entities that are irrelevant to the extraction task, so these were removed. We also created a number of new entity types in support of our extraction task. Below is the taxonomy for the entities in our system.

```
- Entity:
    # Any BioEntity may appear as the controlled in an Activation
    - BioEntity:
        - Virus:
            - CommonVector
        - Vaccine:
            - SubunitVaccine
        - BioProcess   # ex. "apoptosis"
        - BioChemicalEntity:
            - Generic_entity
            - Simple_chemical
            - Equivalable:
                - Family
                - MacroMolecule:
                    - Protein
                    - Gene_or_gene_product
                    - Complex
                    - GENE
    - VaccineType:
          - DNAVaccineType
          - RNAVaccineType
          - SubunitVaccineType
          - InactivatedVaccineType
          - VectorVaccineType
          - LiveVaccineType
```

Some of these are entities similar to those already existing in REACH - for instance, the virus and vaccine types. Others, such as the `VaccineType` are used in rules used to extract relationships.

Writing Rules

There are two different rule types for extracting entities. The first uses the named entity recognition (NER) system within the `processors` Scala package. The second utilizes surface and syntax rules in REACH. Steps needed to prepare the system and example rules are described below.

Named Entity Recognition

The biological databases that contain pathways, proteins, etc. are in the `bioresources` Scala package referenced from REACH. First, we generated text files with the names of entities the system should use for NER, along with database identifiers, if applicable. For example, a subset of virus names in the `ImportantVirus.tsv.gz` file are shown below. Here, the database identifiers are from the NCBI taxonomy.

```
Chikungunya 37124
Chikungunya Virus   37124
CHIKV   37124
Coxsackievirus B3   12072
coxsackievirus B3 CVB3  12072
Coxsackievirus B3 virus 12072
Coxsackievirus virus B3 12072
```

Once the dictionary had been created, the `ner_kb.config` file was updated with the new information and the files used by the `processors` library was updated with the provided script. In this case, the following line, which states that the `ImportantVirus.tsv.gz` file contains entities of type `Virus` is added

```
ImportantVirus  Virus
```

REACH references the `processors` and `bioresources` library versions from SBT, so in order to get the system to see this updated version, a new version number was created and `deploy-local` was used to install that so that SBT could "see" the correct version. There may be better ways to do this, but this was suggested by a coworker with more experience in Scala who ran into a similar problem. This needed to be done with both the libraries. Once these libraries were deployed, the REACH `build.sbt` file was changed to reference the new versions.

The rules for the NER extraction take the same format as those already in REACH. For example, here is the rule for viruses.

```
- name: ner-virus
  label: Virus
  action: mkNERMentions
  priority: 3
  type: token
  pattern: |
    [entity='B-Virus'] [entity='I-Virus']*
```

Syntax or Surface Rules

We also employed REACH rules to identify specific types of entities without using NER. For example, we created rules to identify vaccine types. In the simplest case, a rule that captures "live [vaccine]" or "live attenuated [vaccine]" is shown below.

```
- name: live-liveattoptional
  example: "Live (attenuated) vaccine"
  label: LiveVaccineType
  type: token
  action: mkBioMention
  pattern: |
    [lemma=live] [lemma=attenuated]? (?= []* [mention=Vaccine])
```

Adding support for Grounding

Grounding, or the assignment of database identifiers is one of the functions REACH adds on top of ODIN, the rule-based extraction system it is built on top of. In order to get our new entities to be grounded correctly, we had to make changes to the following classes

ReachKBConstants

Here, for consistency with the rest of REACH, we added constants for the filenames containing the databases we created in the NER step. For example,

```
val StaticVirusFilename = "ImportantVirus.tsv.gz"
```

ReachIMKBLookups

Here we created a lookup for each of the entity types that needed grounding. So, for the Virus entity, the lookup was defined as:

```
  /** KB lookup for viruses -> points to NCBI taxonomy **/
  def staticVirusKBLookup: IMKBLookup = {
    val metaInfo = new IMKBMetaInfo(
      kbFilename = Some(StaticVirusFilename),
      namespace = "virus_ncbi",
      baseURI = "http://identifiers.org/taxonomy/",
      resourceId = "MIR:00000006"
    )
    new IMKBLookup(TsvIMKBFactory.make(metaInfo))
  }
```

We also added this lookup to the list of singletons at the top of the file

```
val StaticVirus = staticVirusKBLookup
```

ReachIMKBMentionLookups

Similarly, we updated this file to add the accessor for each of the entity types.

```
  /** KB accessor for viruses using static KB **/
  def staticVirusKBML: IMKBMentionLookup = {
    val metaInfo = new IMKBMetaInfo(
      kbFilename = Some(StaticVirusFilename),
      namespace = "virus_ncbi",
      baseURI = "http://identifiers.org/taxonomy/",
      resourceId = "MIR:00000006"
    )
    new IMKBMentionLookup(TsvIMKBFactory.make(metaInfo))
  }
```

Running Reading Pipeline

Merging REACH and Custom Rules

In order to incorporate our custom rules with some of the existing rules, we created a separate package to store our custom grammar, and wrote a RuleReader to reference these custom rules. This amounted to changing the *resourcesPath* variable. To combine REACH and custom rules, we imported REACH rules in custom `entities_master.yml` and `events_master.yml` files, as shown below

```
taxonomy: org/mitre/grammar/taxonomy.yml

rules:
  - import: org/mitre/grammar/entities/virus_entities.yml
  - import: org/mitre/grammar/entities/trial_indicators.yml
  - import: org/mitre/grammar/entities/time_expressions.yml
  - import: org/mitre/grammar/entities/human_indicators.yml
  - import: org/mitre/grammar/entities/persistence_indicators.yml
  - import: org/mitre/grammar/entities/immune_indicators.yml
  - import: org/clulab/reach/biogrammar/entities/entities.yml
  - import: org/clulab/reach/biogrammar/entities/mutants.yml
  - import: org/clulab/reach/biogrammar/coref/generic_entities.yml
    vars:
      earlyPriority: "5"
      priority: "7"
```

Modifying Reading Pipeline

REACH implements some complex logic for dealing with partial events, as well as co-reference when reading for molecular mechanism. When we first added our rules to the REACH pipeline, we noticed that the output from our rules was getting filtered out. So, in our copies of the `ReachSystem` class, we made the following minor changes

- Removed logic around modification rules, since they are not used in our reading pipeline
- Changed PaperReader to reference the custom ReachSystem class

Output of Reading Results

The majority of our modifications to the platform focused on handling system output. While the FRIES format used for Big Mechanism was appropriate for our purposes, when we added rules for our new event and relationship types, REACH's `JsonOutputter` class was throwing Exceptions. This required a couple of modifications to the code. The original approach was to create a sub-class, but because some of the functionality was in an Object, we were unable to extend this.

Handling New Event Types

One of the limitations we ran into was that the JSON output was limited to a set of hard-coded event types. So, as we added these, we needed to update the event types. For example, to prevent an error when processing the VirusEntryReceptor event type, we had to add the following to `mkEventType`, which generates the event type string

```
if (label == "VirusEntryReceptor")
  return "virus-entry-receptor"
```

We also used ODIN relationships, rather than events, for some of the information extraction types. These also caused errors in output. To handle this, we had to make sure Relationship mentions were captured in the `mkArgType` function:

```
else if (arg matches "Relationship") "event"
```

This allowed us to use the FRIES format for our downstream tasks without much additional engineering.