decheem Documentation

Release latest

Apr 14, 2019

User Documentation

CHAPTER 1

What is deCheem?

deCheem is a logic programming framework that makes use of a declarative programming interface/language. It is designed to facilitate deductive thinking in business and philosophy, where juggling numerous overlapping statements and facts is required to reach a conclusion.

It can be seen as a cross-over between a method of automating Socratic sessions and a "pivot table" solution that works with unnormalised data. Another way to describe it would be to call it a method for "querying SQL queries". Others might call it Venn diagrams for situations too complex to visualise.

deCheem databases consist of assertions about the world based on conditionals. The assertions excludes or includes certain situations in the world, which allows for conclusions to be drawn by iteration, starting from certain boundary conditions.

Please click on the Intro section below to learn more about the theory behind deCheem:

1.1 Introduction to deCheem

1.1.1 The purpose of deCheem

Human language can be amazingly nuanced, allowing us to communicate complicated concepts that are specific to a certain context, culture or emotional state.

However, this very property of human language starts to limit us when we start taking on extremely complicated discussions involving multiple conflicting arguments and beliefs. Examples of this manifesting in our daily lives are plenty: business meetings that drag on for hours in search of a best-fit solution when actually none is possible if all the requirements are properly understood. People spending their lives chasing certain ideals and dreams, unaware of how these are at odds with their fundamental beliefs and values.

Most of the time, people involved in these modern-day 'Socratic sessions' have more than enough brain power to understand the arguments in the discussion when taken one at a time. However, when several arguments need to be stringed together to reach conclusions that are not necessarily intuitive, people start to struggle.

Often it is because the arguments are part of dense and lengthy essays or dialogues, making it difficult for one to keep remember and thus comprehend all it's components holistically. Other times, the conclusions are reached but

dismissed, because they are so unintuitive that one simply assumes that his or her reasoning must have been faulty.

deCheem's was born out of the need for a method to express arguments, beliefs and statements in a consistent and succinct manner, so that watertight reasoning can be performed based on them. With it, people gain access to a robust framework for conducting discourse and argumentation and cut down the confusion and misunderstandings so common in our daily lives.

1.1.2 The deCheemic way of describing things

deCheem is basically a multi-dimensional Venn diagram. It's purpose is to help us see the world around us as a 'universal set', and then framing normal human sentences as statements about the sets in this world.

To understand how deCheem works, it is important to learn to frame statements and arguments in a deCheemic way.

In the deCheem framework, all that we say, write and think about the world can be said to comprise of two parts:

- Description of what that thing is, which can be a compound statement consisting of multiple descriptive properties.
- A modal, which is a normative or prescriptive component of our thoughts about the thing (or category of thing) described.

Consider the sentences below as examples:

- I don't like to eat fish.
- Things that are sour are not nice to eat

In deCheemic terms, this can be broken down into:

descriptive properties	Modal
Fish	don't like to be eaten by me.
Things that are sour	are not nice to eat

As it is possible to have a mental image of anything that we are able to imagine, and anything that we can imagine can eventually be described by some form of language or another, there should not be any limits to what can be described under deCheem, assuming the right number of descriptive properties with the right amount of descriptive specificity are used.

1.1.3 Theory behind deCheem: the concept of All Possible Combinations

Before we talk about deCheem's syntax and functioning, it is important to understand the thought framework that underlies it, which is based on set-theory.

Imagine there is some called Jan who holds the following three simple beliefs (described with placeholder adjectives):

- Things which are 'KAKA' exist.
- Things which are 'ALPHA' and 'GAMMA' do not exist.
- Things which are 'ALPHA' and 'YETA', but not 'BETA', do not exist.

Let us then express his beliefs in the following deCheemic form:

```
          descriptive_properties          | statementmodal
------------------------------------------+----------------
 KAKA=TRUE                                | EXISTS
 ALPHA=TRUE AND GAMMA=TRUE                | NOT EXISTS
```

(continues on next page)

(continued from previous page)

```
 ALPHA=FALSE AND BETA=FALSE AND YETA=TRUE | NOT EXISTS
(3 rows)
```

'Things' here simply refer to the most generic 'thing' one can imagine; everything from concepts, ideas, people, animals to objects all fall under 'things'. It is the basic term one can use to describe anything in the universal set of everything imaginable in this universe.

In this situation, ALPHA, KAKA, GAMMA, BETA and YETA are placeholder adjectives, which I shall call **descriptive properties** of things. Once a descriptive property has been described, it opens up the possibility for all things in the universe to be described by it. There can be three ways in which a thing can interact with a descriptive property:

1. be positively described by it (e.g. it is KAKA).
2. be negatively described by it (e.g. it is not KAKA).
3. have no known descriptive relationship with it (e.g. it is not known whether it could be KAKA or not)

To come up with a framework to describe things in case 1 and 2, every descriptive property needs to be able to take up the boolean values of either True or False. To encapsulate the third case, we need to take into account cases where things can either be True or False on every descriptive property.

What this means is that we need to imagine a table of all possible combinations of True/False values for all descriptive properties mentioned. If we extract the descriptive properties from Jan's three statements and flesh out all possible combinations, we will get 32 possible combinations (simply 2 to the power of 5):

```
 rowid | gamma | beta | kaka | yeta | alpha
-------+-------+------+------+------+-------
     1 | t     | t    | t    | t    | t
     2 | t     | t    | t    | t    | f
     3 | t     | t    | t    | f    | t
     4 | t     | t    | t    | f    | f
     5 | t     | t    | f    | t    | t
     6 | t     | t    | f    | t    | f
     7 | t     | t    | f    | f    | t
     8 | t     | t    | f    | f    | f
     9 | t     | f    | t    | t    | t
    10 | t     | f    | t    | t    | f
    11 | t     | f    | t    | f    | t
    12 | t     | f    | t    | f    | f
    13 | t     | f    | f    | t    | t
    14 | t     | f    | f    | t    | f
    15 | t     | f    | f    | f    | t
    16 | t     | f    | f    | f    | f
    17 | f     | t    | t    | t    | t
    18 | f     | t    | t    | t    | f
    19 | f     | t    | t    | f    | t
    20 | f     | t    | t    | f    | f
    21 | f     | t    | f    | t    | t
    22 | f     | t    | f    | t    | f
    23 | f     | t    | f    | f    | t
    24 | f     | t    | f    | f    | f
    25 | f     | f    | t    | t    | t
    26 | f     | f    | t    | t    | f
    27 | f     | f    | t    | f    | t
    28 | f     | f    | t    | f    | f
    29 | f     | f    | f    | t    | t
    30 | f     | f    | f    | t    | f
    31 | f     | f    | f    | f    | t
```

(continues on next page)

(continued from previous page)

```
    32 | f      | f     | f     | f     | f
(32 rows)
```

With this table of all possible combinations, which I shall call the **table of combinations**, we can start using the deCheemic form of Jan's statements of belief as SQL queries to mark the subsets of combinations that he deems to be existent or not.

```
 rowid | gamma | beta | kaka | yeta | alpha |                    statement_modals
-------+-------+------+------+------+-------+-----------------------------------------
↪-------------
     1 | t     | t    | t    | t    | t     | Statement 2 says NOT EXISTS, Statement␣
↪1 says EXISTS
     2 | t     | t    | t    | t    | f     | Statement 1 says EXISTS
     3 | t     | t    | t    | f    | t     | Statement 1 says EXISTS, Statement 2␣
↪says NOT EXISTS
     4 | t     | t    | t    | f    | f     | Statement 1 says EXISTS
     5 | t     | t    | f    | t    | t     | Statement 2 says NOT EXISTS
     6 | t     | t    | f    | t    | f     |
     7 | t     | t    | f    | f    | t     | Statement 2 says NOT EXISTS
     8 | t     | t    | f    | f    | f     |
     9 | t     | f    | t    | t    | t     | Statement 1 says EXISTS, Statement 2␣
↪says NOT EXISTS
    10 | t     | f    | t    | t    | f     | Statement 1 says EXISTS, Statement 3␣
↪says NOT EXISTS
    11 | t     | f    | t    | f    | t     | Statement 1 says EXISTS, Statement 2␣
↪says NOT EXISTS
    12 | t     | f    | t    | f    | f     | Statement 1 says EXISTS
    13 | t     | f    | f    | t    | t     | Statement 2 says NOT EXISTS
    14 | t     | f    | f    | t    | f     | Statement 3 says NOT EXISTS
    15 | t     | f    | f    | f    | t     | Statement 2 says NOT EXISTS
    16 | t     | f    | f    | f    | f     |
    17 | f     | t    | t    | t    | t     | Statement 1 says EXISTS
    18 | f     | t    | t    | t    | f     | Statement 1 says EXISTS
    19 | f     | t    | t    | f    | t     | Statement 1 says EXISTS
    20 | f     | t    | t    | f    | f     | Statement 1 says EXISTS
    21 | f     | t    | f    | t    | t     |
    22 | f     | t    | f    | t    | f     |
    23 | f     | t    | f    | f    | t     |
    24 | f     | t    | f    | f    | f     |
    25 | f     | f    | t    | t    | t     | Statement 1 says EXISTS
    26 | f     | f    | t    | t    | f     | Statement 1 says EXISTS, Statement 3␣
↪says NOT EXISTS
    27 | f     | f    | t    | f    | t     | Statement 1 says EXISTS
    28 | f     | f    | t    | f    | f     | Statement 1 says EXISTS
    29 | f     | f    | f    | t    | t     |
    30 | f     | f    | f    | t    | f     | Statement 3 says NOT EXISTS
    31 | f     | f    | f    | f    | t     |
    32 | f     | f    | f    | f    | f     |
(32 rows)
```

From this marked table of combinations, the stage is set for all kinds of interesting queries to be run, and for linguistic expressions like 'Always', 'Never', 'Sometimes' and 'Possible' to be constructed on top of it.

For example, an interesting query would be to looking for the cases where Jan's statements would contradict each other (e.g. rowids 1,3,9,10,11,26).

However an even more interesting query would be to see what Jan did not explicitly say, but would imply under certain conditions.

Let's say we want to know what 'things' would be acceptable for Jan under the anchoring condition that things need to be GAMMA but not BETA. By filtering on the rows without 'NOT EXISTS' statements and with GAMMA=True and BETA=FAlse, we have two rows remaining.

```
 rowid | gamma | beta | kaka | yeta | alpha |    statement_modals
-------+-------+------+------+------+-------+------------------------
    12 | t     | f    | t    | f    | f     | Statement 1 says EXISTS
    16 | t     | f    | f    | f    | f     |
(2 rows)
```

Summarising these, you will see:

```
 descriptive_property | true_count | false_count
----------------------+------------+-------------
 KAKA                 |          1 |           1
 YETA                 |          0 |           2
 ALPHA                |          0 |           2
(3 rows)
```

This implies that under the condition of things being GAMMA and not being BETA, things will definitely need to **not** be YETA or ALPHA. There is however no verdict on things having to be KAKA or not under these circumstances, which is in line with Jan's beliefs.

1.2 Workflow and elements in a deCheemic analysis

The primary use case of deCheem is to help people discover the implications of a particular set of view points on other topics within a certain world view.

For example, given a South American society's world view as a background, deCheem will allow you to find out what one's strong belief in meritocracy in such an environment would mean for his/her views on the natural behaviour of people (aka state of nature), government, society and religion.

1.2.1 State-Categorise-Inquire - the deCheem workflow

deCheem statements are the building blocks of any deCheem analysis, and are used to construct statements of belief, categorisation and inquiry.

Each statement has three main components and a predicate 'be':

1. **Type **

- Can take the types of **IF**, **ELIF**, **THEN** or **ASK**
  - The **IF** and **ELIF** types simply checks if the statement is true for what it claims given the current world view.
  - The **THEN** type asserts a certain statement and makes that part of the world view.
  - The **ASK** type doesn't assert, and simply queries what the preceding THEN statements state to be the implications for the topic/subject/property that is being **ASK** -ed.

2. **Subject Phrases**

- Sets the subjects(s) of the statesment

3. **Modal Phrases**

- Sets the adjectival conditions that we want to check or assert on the topic. * Adjectival conditions can take three forms:
  - **Always**
  - **Possibly**
  - **Never**

Consider the following statement:

```
Societies that are secular and socialist are never capitalist and always egalitarian.
```

This statement can then be expressed in the deCheem JSON syntax as:

```
-case: LET (is_liberty) be Always (is_claimed) # Liberty is always claimed.
```

Statements can also be stringed together using conditionals to create more complex cross-topic statements:

```
- case: LET (is_liberty,is_claimed) be Always (restricted) # Liberty that's claimed␣
↪is always restricted in nature
- case: IF (is_liberty) be Always (restricted) # If liberty is restricted....
  then:
  - case: LET (is_emancipation) be Never (is_human_emancipation) #then emancipation␣
↪will never be true human emancipation
```

As mentioned below, the State-Categorise-Inquire workflow requires three collections of statements that need to be developed in parallel:

1. **Statements of beliefs** - (conditional) statements indicating a certain belief, which sets the stage for the triggering of conditional statements.
2. **Statements of conditionals** - IF and ELIF statements that get triggered based on what background conditions we have declared in the first section.
3. **Statements of inquiry** - these are strictly ASK statements, which serve as boundary conditions to query the implications of a particular set of belief statements for a certain topic.

The interactions of these three sets of statements will be illustrated in the next section on examples.

1.3 Quick-start deCheemQL Example

Often we come across situations where there are many rules, restrictions and facts that we need to compare with each other in order to. That is where deCheem comes into play.

Each deCheemQL statement consists only of 4 simple parts:

1. **Statement type** - this starts off the sentence and can be either 'LET', 'IF', 'ELIF' or 'ASK'
2. **Subject phrases** - the adjectives that describe a subject are typed out (separated by commas), and enclosed in brackets. Multiple subjects can be stringed together by 'and' (e.g. (is_apple,is_green) and (is_orange,is_yellow_red)). Negate an adjective by prepending a '!' sign.
3. **The 'be' word** - this is to indicate the preceding subjects 'being' the modal phrases after that
4. **Modal phrases**- these are similar to subject phrases, but indicate whether a set of adjectives are always, never or possibly true. Preceed every phrase with either 'Always', 'Never' or 'Possibly'.

deCheemQL is basically **YAML** with some deCheemQL specifics. Each statement is started off with a '- case' line header, followed by the deCheemQL statement: .. code-block:: yaml

- case: LET (is_apple) and (is_orange) be Always(sweet,delicious) and Never(smelly,spicy)

Conditional statements like 'IF' and 'ELIF' can have a child 'then' statement under it, and can be nested as many layers as you need, in any order or combination you like.

```
then:
- SECTION_HEADER: &statements # We use '&' to denote a YAML anchor, which Sublime␣
↪Text recognises as a symbol and thus useful for toggling between sections of the␣
↪statements with 'Command+R'.
- case: LET (is_apple) and (is_orange) be Always(sweet,delicious) and Never(smelly,
↪spicy)
- case: IF (is_apple) be Never(sour)
  - case: LET (is_cake) be Always (tarty,creamy)
- case: ELIF (is_apple) be Never(smelly,spicy) #Needs to be preceded by a IF or ELIF␣
↪case at the same level just before
  then:
  - case: LET (is_desert) be Always (raining,thundering)
- case: ASK (is_sky)
- case: ASK (is_cake)
```

1.3.1 Test it out yourself!

The easiest way to test out deCheem is to use the deCheem online sandbox that allows you to test out deCheem in real-time.

However, the best way to get started is to use Sublime Text as your YAML editor, and using the RESTer package to send HTTP requests to the free and open deCheem processor endpoint right from your editor!

Simply copy-paste the example below into Sublime Text and execute the entire file with the *RESTer: HTTP Request* command. The results will then be displayed in a tab right next to the input file.

```
POST http://gmkung.pythonanywhere.com/yaml/
@response_group: 1
@response_group_clean: true
@request_focus: true

---
then:
#Let this be the situation.
- case: LET (is_human_right) be Always(necessary)

#If he says ...
- case: IF (is_human_right) be Always (necessary) and Possibly (compromised)
  then:
  - case: LET (is_military_action) be Always(justified) and Possibly (!state_financed)
  - case: LET (is_human_right) be Always (derived_from_religion)

# and she says ...
- case: IF (is_human_right) be Possibly (derived_from_religion)
  then:
  - case: IF (god) be Possibly (existent)
    then:
    - case: LET (is_religion) be Always (state_financed)

# and I say ...
- case: IF (state_financed) be Possibly(!existent) and Possibly (!holy)
  then:
```

(continues on next page)

(continued from previous page)

```
  - case: LET (is_citizen) be Never (tax_paying)

# ... then let us find out what we all say about the citizen, military action,␣
↪religion and is_human_rights
# based on the assumption that human rights are necessary
- case: ASK (is_citizen)
- case: ASK (is_military_action)
- case: ASK (is_religion)
- case: ASK (is_human_right)

#The following are just parameters for the analysis, so copy them to the file, but don
↪'t dive into their meaning for now!
apiVersion: deCheem4.0
verboseResults: false
responseFormat: yaml
showPossibilities: false
responseFormat: yaml
AlwaysTerms: does
PossiblyTerms:  Perhaps
NeverTerms: impossible
humanFriendlyOutput: true
```

The YAML response that comes back will tell you what it means to be a 'sky' or 'cake' based on the beliefs in this set:

```
- collectionName: DefaultCollection
  consequenceResults:
  - then: []
  - then:
    - then: []
  - then: []
  - newTotalBoundary: {is_citizen: true, tax_paying: false}
    originalBoundary: {is_citizen: true}
    possible: true
  - newTotalBoundary: {is_military_action: true, justified: true}
    originalBoundary: {is_military_action: true}
    possible: true
  - newTotalBoundary: {is_religion: true, state_financed: true}
    originalBoundary: {is_religion: true}
    possible: true
  - newTotalBoundary: {derived_from_religion: true, is_human_right: true, necessary:␣
↪true}
    originalBoundary: {is_human_right: true}
    possible: true
  manifestationCollections:
  - collectionName: null
    manifestationCollections: []
```

1.4 Frequently Asked Questions

- Why is it so that you cannot have compound/layered boundary conditions in a deCheem session?
- From a linguistic point of view, boundary conditions need to be consistent within itself. We should therefore not have contradictory filtering properties within a single boundary condition (e.g. is_cat=true AND is_dog=true), as the results would not make sense unless we are explicitly testing for the possibility of hybrid situations. If

there are multiple boundary conditions, they should be tested in parallel instead of in serial, as it will inevitably cause inconsistent or impossible situations.

- Why can conditional-assertion statements be layered then?
- These statements are meant to filter the table of all possibilities, and layering them simply means defining ways in which different versions of this filtered table can be constructed.
- Why is it important to take note of the sequence of the conditional-assertion statements?
- deCheem is meant to help us define situations in the world, and situations are derived from a certain history, which is basically a sequence of events.
- Where should the boundary conditionals be triggered?
- To get clarity on how to handle them, we need to break them down into its components, the filters and the assertions. The assertions of Always, Possible or Never can be included at any point in the world creation process. But the filters can also be inserted any point in the process, or even at the end, as filters do not trigger conditionals (only assertions do). Boundary conditions can be seen as verbose printing of the implications of only the filter of any statement, and if we want verbose printing for the sake of finding out implications for certain boundary filters, we should use only the IF and ELIF types to avoid unintentional additional assertions.

1.5 Get in touch

deCheem is a project by Guangmian Kung, and is currently implemented in two form:

1. a straightforward Python script with a PostgreSQL backup that generateds 2^N rows as described in the section on Table of Combinations. This implementation allows for easy understanding of the results, but does not scale beyond 20 descriptive properties.
2. a streamlined version - also Python+PostgreSQL based - that has been tested for 100k descriptive properties and above, and also implements 'conditionals' allowing for statements describing totally different subjects to have dependencies on each other. This version is however less easily queryable. The section 'Example applications of deCheem' makes use of this implementation.

As effective use of deCheem requires guidance and consultation, please reach out to Guangmian Kung at guangmian@gmail.com to find out how to test deCheem out.