Concepts for URL Shortening

1. The NonceGeneration concept ensures that the short strings it generates will be unique and not result in conflicts. What are the contexts for, and what will a context end up being in the URL shortening app?

A context is used to organize the set of random strings that are associated with the specific context. A context in the URL shortening app would be the base part of the URL. The suffix part of the URL would be the randomly generated short string.

2. Why must the NonceGeneration store sets of used strings? One simple way to implement the NonceGeneration is to maintain a counter for each context and increment it every time the generate action is called. In this case, how is the set of used strings in the specification related to the counter in the implementation? (In abstract data type lingo, this is asking you to describe an abstraction function.)

The size of the set of used strings will be equal to the counter for the associated context. NonceGeneration must store a set of strings because every time a string is generated, the generator must verify that it is unique within that context. A simple counter will not be able to achieve this same functionality.

3. One option for nonce generation is to use common dictionary words (in the style of yellkey.com, for example) resulting in more easily remembered shortenings. What is one advantage and one disadvantage of this scheme, both from the perspective of the user? How would you modify the NonceGeneration concept to realize this idea?

One advantage of using common dictionary words is that the URLs will be much easier to type and share. One disadvantage though, is that there is a possibility that these randomly generated words could result in something offensive to the user. The generate action of NonceGeneration would need to be changed by choosing from a set of common words to return as the Nonce. This result would still need to be checked to make sure it is unique.

Synchronization Questions

1. In the first sync (called generate), the Request.shortenUrl action in the when clause includes the shortUrlBase argument but not the targetUrl argument. In the second sync (called register) both appear. Why is this?

The first sync requires the shortUrlBase because the when clause, “generate,” requires the base in order to generate the Nonce. The second sync requires the targetUrl because this is the step where the generated URL is associated with the original one. These syncs are not given the arguments until they are needed in the when clause.

2. The convention that allows names to be omitted when argument or result names are the same as their variable names is convenient and allows for a more succinct specification. Why isn't this convention used in every case?

One example when this can't be done is when an argument did not come as a variable, like the seconds value in the third sync. Another example is when an action returns a value by a specific name; it would be confusing to change the name of that result just for a more succinct specification.

3. Why is the request action included in the first two syncs but not the third one?

The request action is not included in the third sync because this sync is done automatically after registration and does not request any new user input.

4. Suppose the application did not support alternative domain names, and always used a fixed one such as "bit.ly." How would you change the synchronizations to implement this?

This would remove the need for the variable "shortUrlBase" since the base would always be the same. This could remove the need for the first sync, as this action could be combined with the second sync.

5. These synchronizations are not complete; in particular, they don't do anything when a resource expires. Write a sync for this case, using appropriate actions from the ExpiringResource and URLShortening concepts.

sync deleteUrl
when ExpiringResource.expireResource (): (resource)
then UrlShortening.delete (resource)

<u>**Extending the design**</u>

concept CounterCreation [Resource]
purpose to initialize the counter for the resource
principle a counter will be created for the given resources
state
    a set of resources
        with a counter
actions
  createCounter (resource: Resource) : ()
    effect creates a counter associated with the given resource

```
concept Counter
purpose counter meant to hold number of times a URL was accessed
principle on each access, the counter will increase by one
          shows count when requested
state
    a set of counter
        a Resource
        a number
actions
showCount (resource: Resource, access: Boolean): (count: Number)
    effect returns the count of the counter associated with the given resource if access is True
           otherwise returns an access error message
system incrementCounter (resource: Resource) : ()
    requires resource is associated with a counter
    effect increments the number of the counter associated with the resource

concept UserVerification [Resource]
purpose to verify identify of users examining analytics
principle on resource creation a user will be associated with it
          on attempt to access analytics, user will be verified
state
   a set of Resources
        a user String
actions
registerResource (resource: Resource, user: String): ()
   effect permanently associates the given resource with the given user
analyzeAnalytics (resource: Resource, user: String): (access: Boolean)
   effect verifies if the given user matches the given resource

sync creation
 when UrlShortening.register (): (shortUrl)
 then UserVerification.registerResource(shortUrl, user)
      CounterCreation.createCounter (Resource: shortUrl)

sync translations
 when Request.lookup (shortUrl): (targetUrl)
 then Counter.incrementCounter (shortUrl)

sync examineAnalytics
 when Request.analyzeAnalytics (shortUrl, user): (access)
 then Counter.showCount (shortUrl, access): (result: Number)
```

Allowing users to choose their own short URLs - This would be a great feature and could be implemented by first changing the NonceGeneration concept to be able to verify if an input string is already associated with a concept. After this change, the register sync can be altered to accept a user input string that will be verified against other strings in the context. The rest of the process can function similarly, with the only change being that the nonce will be replaced with the verified user-input string.

Using the “word as nonce” strategy to generate more memorable short URLs - I believe this feature would be undesirable for various reasons. One of the reasons is that there would be significantly fewer variations of short URLs if we were using common words as a nonce. Another reason is that there is an increased chance of an offensive word or phrase being generated by the concept.

Including the target URL in analytics, so that lookups of different short URLs can be grouped together when they refer to the same target URL - This additional feature could be implemented by changing the syncs dealing with analytics. The only change that would need to be made in my implementation is to pass the target URL as the resource to the new concepts instead of the short URL.

Generate short URLs that are not easily guessed - This is a great, simple change that can be made by altering the NonceGeneration concept. The generate action in this concept would need to be updated, adding a clause that would add some requirements to the nonce generated. For example, these requirements could be that the nonce has numbers or allowed symbols mixed in with letters.

Supporting reporting of analytics to creators of short URLs who have not registered as user - I do not believe this would be a desirable feature, as this could lead to a breach of privacy. If a creator of a short URL does not register as a user, there is likely no need for them to do more advanced tasks, such as seeing analytics reports. It would be much safer and easier to keep track of analytics if they had an associated user with them and their short URL.