IMPACT: International Journal of Research in Engineering & Technology (IMPACT: IJRET)
ISSN(E): 2321-8843; ISSN(P): 2347-4599
Vol. 3, Issue 7, Jul 2015, 37 48

SEMANTIC VERSIONING IN ONTOLOGIES

AASTHA MAHAJAN[1] & PARMINDER KAUR[2]

[1]Research Student, Department of Computer Science & Engineering, Guru Nanak Dev University, Punjab, India

[2]Assistant Professor, Department of Computer Science & Engineering., Guru Nanak Dev University, Punjab, India

ABSTRACT

This paper gives you an idea concerning semantic versioning pertaining to semantically rich ontlogies. As SemVer is being adopted worldwide for versioning different software systems, it makes dependency hell a thing of the past and solves two problems of version lock and version promiscuity. This paper reveals the importance of associating well defined semantics with the versioning schemes to make users, either providers or consumers of the software clear about the dependency/compatibility issues. In this paper, the importance of SemVer and different versioning schemes has been conversed. In the end, semantic versioning is applied to ontology based information system. Cooking domain ontology is created in protégé and new changes are induced, as a result, new versions are likely to form. The kind of changes made could be judged through SemVer scheme based on, which part of the scheme is changed, owing to the changes affecting the ontology consumers and providers. The most important thing is to declare the ontology as a public API in order to use semantic versioning otherwise associating meaning with the versioning scheme would be of no use.

KEYWORDS: API, Compatibility, Cooking, Dependency, Domain, Eclipse, Ontology, OSGi, Protégé, SemVer, Semantics, Versioning

INTRODUCTION

Semantics are anything that describes the work of an art to bring out its essence, thus, making it more understandable. Semantics means a strict, clearly defined meaning. In the domain of computer science, semantic builds relationship between syntax and its interpretation [1].In the last decade, efforts have been made to associate semantics with the data , giving rise to intelligent systems. These systems revolutionize due to changes in specifications, conceptualizations or requirements. The amendments induced results in the creation of new versions which may or may not be backward compatible with each other. Each new version is designated an URI (Uniform Resource Identifier) which is inimitable for each version. The version number associated with each version confers some meaning to it. Semantic Versioning is the term used when meaning is associated with the versioning scheme. In reality, not everything is backward compatible [3].Semantic versioning, SemVer for short is now being adopted worldwide for versioning different software systems. A semantic version is a change with the strict meaning. Without semantic versions, the importer and the exporter of the API (Application Programming Interface) have no way of communicating backward compatibility and incompatibility[3]. Semantic versioning uses three or four numbers for versioning the software systems instead of two. The most important requirement for using semantic versioning is declaring the software as a public API. Public API may consist of documentation or be enforced by the code itself. Declaring the software as a public API in semantic versioning is of chief magnitude because it let the user to know what kind of changes have been made to the API, assigning meaning otherwise would not be useful if the of piece of software is not declared publicly. For instance, suppose a piece of software

has version 6.2.5, in semantic versioning, each part conveys some meaning.

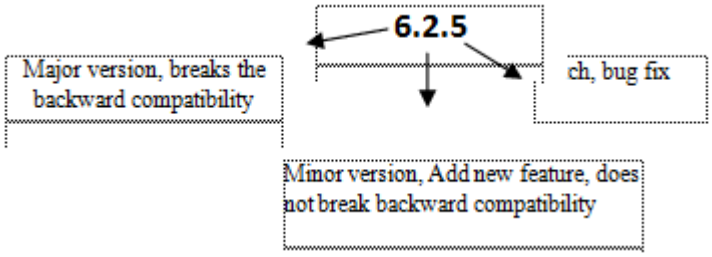

Figure 1: Shows Three Segments of Semver Scheme

Semantic versioning scheme consists of three parts Major.Minor.Patch. Each part conveys some connotation regarding the changes made to the API . In the above example, 6 is the major version number, Major number indicates the number of changes made to the API that were not backward compatible since the version release. Deleting a message in an API is a major version change because it breaks both the consumers and providers of the API. Everything needs to be recompiled. Therefore, the version 6 of the piece of software is not backward compatible with the version 5 as it breaks the API. Minor version number indicates the change that is backward compatible and therefore does not break the API. It means a new feature has been added in the API. To show its addition, an increment is made in the minor version number. The new feature addition breaks the provider of the API, but do not break the consumer of the API, consumers are backward compatible(OSGi versioning scheme differentiates between providers and consumers unlike other versioning schemes). The provider needs to implement this new change and API needs to be recompiled, therefore , it breaks the API for providers, but consumers does not necessarily have to use it. The last number is the patch number, it is incremented each time bug fixes are done like security updates etc and it never breaks the API. In case, a security update breaks the API , this end up in creating a whole new version by incrementing the major version number. In the example, the piece of software is at the version number 6.2.5, there have been 5 changes that have broken backward compatibility in the past. For the major version number 6, 2 new features have been added, indicated by the minor version number. The version number 6.2 is backward compatible with the version number 6.1. The number of bug fixes for the version number 6.2 is 5. Suppose, a new feature is added to the version 6.2.5, the next version of the piece of software is 6.3.0. The patch number is reset to 0, as no bug fixes are done yet for the version number 6.3. When a major change is completed, a switch is made to the new version with version number 7.0.0.

This paper is organized as follows, section 2 emphasize on the importance of semantic versioning, why it is gaining popularity. An example is given to illustrate the same. Section 3 covers different schemes for creating semantic versioned versions- Eclipse versioning schemes and OSGi versioning scheme are the two schemes. Primarily, semantic versioning creates a version composed of three segments and fourth segment is optional. In section 4, an attempt has been made to illustrate how semantically rich ontologies can be semantically versioned, what kind of changes affect which segment of semantic versioning has been discussed. Section 5 concludes this work

Why to Bring Into Play Semantic Versioning?

Now that it has been shown what exactly semantic versioning means, looking frontward some benefits of semantic versioning are outlined. In the world of software , there exists a dread place called “ dependency hell ”[2].Unlike traditional versioning schemes SemVer uses three numbers or sometimes four numbers (Major.Minor.Patch.Qualifier, here

qualifier is a string associated with each version release like, alpha1,beta,rc2 etc.) which uses two numbers for versioning a piece of software. The idea of using SemVer is not a new idea, but the versioning schemes prior to this were essentially useless for dependency management. As the systems revolutionize, dependent artifacts also need to get upgraded to avoid the danger of version lock. By giving a name and clear definition to the version numbers, it becomes easy to communicate the intentions to the users of the software [2]. Once these intentions are clear, flexible(but not too flexible) dependency specifications can finally be made[2]. One of the most important benefit of SemVer than the previous versioning schemes is, it can make the "dependency hell" a thing of the past. An example will demonstrate that how SemVer can help in avoiding the dependency hell.

Suppose there is a library, called "EnvironmentChange". It requires a package called " GlobalWarming" which is a semantically versioned package. When the library EnvironmentChange was created, GlobalWarming was at the version number 5.1.0. The library EnvironmentChange uses some functionality that was first introduced in 5.1.0, safer way is to specify the dependency of GlobalWarming as greater than or equal to 5.1.0 but less than 6.0.0.Now, when the new version number 5.1.1 and 5.2.0 become available, users know that these versions would be backward compatible with the existing dependent software. As soon as the version 6.0.0 comes out, users would know that their dependency probably been broken and they need to upgrade their software.

Some of the benefits of using Semantic Versioning are as follows:

- Clearer understanding of compatibility/dependencies

When there are dependency problems, especially when the code is dependent on another code, if the code changes, it affects every dependent artifact. With the version number, people can decide whether to upgrade their piece of software or not. If the change made is the major change that is it breaks the API, then the users know that their compatibility dependency is probably been broken.

- Encourage well defined APIs

In order to know whether the piece of software is backward compatible or not, one should have clear idea what the API is doing and how people are using it so that the changes made are appropriate.

- Make upgrade decisions easier

SemVer, makes the upgrade decisions easier, for example, a bug fix is done and a new version is created. Users can decide whether to upgrade their version or not. Even if they don't, it won't break the API. Similarly, when a new feature is added, it too can affect the decision to upgrade the piece of software. When the major change is made, it sends the clear message to the users that they are now not backward compatible with the new release of the software. So , they need to switch to the new software version.

Semantic Versioning Schemes

There are three semantic versioning schemes used to define version number with a clear and a strict meaning. These versioning schemes are as follows:

OSGI (Open Source Gateway Intiative)[3]

The foundation mechanisms provided by OSGi specifications are the version and version range[1].OSGi versioning scheme differentiates between providers and consumers of the API. A version consists of maximum 4 parts: major, minor, micro and qualifier. It also specifies version range to know dependency issues more accurately with the use of square brackets('[` , ']' ), indicating inclusive and parentheses( '(`, ')') indicating exclusive. OSGi scheme specifies that there are some semantic changes that must be handled by a provider to credit the change in the API contract while many of these changes are backward compatible with the consumers. Therefore, OSGi emphasis on different provider and consumer version range so that they could describe their different import requirements on the exporter. Look at the following set of version ranges communicating with the use of semantics.

Exported version 2.1.3 built, the following range provides the given meaning.

[2.1.3) Consumer importer policy, breaks the API beyond version 3 or later for consumers

[2.2.2.3) Provider importer policy, breaks the provider when exporter goes beyond 2.3 or later.

[2.1.3.2.1.4) Strict importer policy: only accepts exporter of version 2.1.3.

1See the sections 3.2.5 and 3.2.6 of the OSGi core specification v4.2.

Eclipse *(*Http://Wiki.Eclipse.Org/Version_Numbering)

Ineclipse versions are composed of four segments, 3integers and a string respectively named Major. Minor. Service. Qualifier Each segment captures a different meaning: Major version indicates a breakage in the API, Minor version indicates externally visible changes, Service indicates bug fixes and changes in the development stream and Qualifier indicates a particular build. Unlike, OSGi it does not differentiate between consumers and providers of the API. When major version is increased minor and service parts are reset to 0.

In both eclipse and OSGi, versioning problems are solved. Eclipse versioning scheme is older versioning scheme than semantic versioning which gained popularity recently. Both seems to be similar as both uses four segments in their versioning scheme but it is not quite the same. The disparity in both the schemes lies in the fourth fragment i.e. qualifier, the dissimilarity is the syntax with the qualifier/metadata information in the last segments of the versions. A qualifier is estranged by a dot and can be almost any string. Compared to the qualifier, the prerelease ID and build metadata are estranged by '-'/'+' and carry a bit more semantic information [5].

An Eclipse developer might sight this as just another qualifier, but be alert that the specification of a prerelease ID and build metadata in fact tells more about the inside of a version than just "any string". One can let oneself instigate for choosing one's qualifiers, but in the Eclipse world, one'll always have the dilemma, that 2.0.0.beta > 2.0.0 [5].

Semantic Versioning in Ontologies (Weblog.Clarkparsia.Com/2011/09/Semantic Versioning)

Ontologies are comparable to public API for data. It could be better defined as a public, machine understandable contract between producers and consumers concerning the meaning of data. There are certain implications in regarding ontology as a public API. The focal implication is that OWL(Ontology Web Language) ontologies should be versioned in analogous to APIs. The question arises here how one can think of versioning ontologies semantically. One solution is to

treat ontology as an API and determining how to apply Semantic Versioning to the task of versioning the ontology as an ontology. The statement , OWL ontologies are semantically versioned, means two things

- Making ontology's version identifier structured and meaningful, i.e. which means encoding some meaning in the string of characters that makes up the version identifier;
- Changing the version identifier according to some well-understood public , and reasonable rules.

The two aforesaid things suggests that a version identifier and rules for changing the version identifiers, is a simple informative mechanism intended to make multi-party coordination involvement cheaper and less disruptive. Semantic Versioning (SemVer) rules can't be directly to OWL ontologies: since they are not actually APIs, therefore, some adaptation is required.

Semantically Versioning OWL (*ontology web language*) ontologies [4]

Following given are the steps that are need to be followed in order to version semantically rich ontologies using SemVer scheme:

- To use SemVer, OWL ontology is declared as public API.
- A version identifier "X.Y.Z" where X, Y, Z are the integers which are incremented. X is the major field; Y is the minor field; and Z is the patch field. Semantic versioning means declaring public rules and conditions for when each field is incremented, based on the impact in OWL ontology.
- Special version identifier, for say, beta releases or release candidates, etc. by affixing some alphanumeric stuff to the patch field.
- Never change an ontology without changing its version identifier. Ontologies are indifferent to code in this respect as any change in version identifier conveys consumers about what is in the new version. If the ontology changes publicly, the version identifier must change too.
- Using a 0 in the major field before the ontology is finalized, that indicates things may change at any time.
- After the ontology is finalized, version identifier 1.0.0 is given which defines the public ontology. Thereafter, an increments in the different fields signals the kind of changes that have been undertaken for the different revisions.

The question that arises here is, what conditions or changes forms the basis for changes to major, minor or build fields. These are hard to decide as ontology is different from that of programming language APIs in some sense. So, the versioning scheme in broader sense regarding consumers could be: if there is a patch change, that version could be ignored, if there is major change that version could not be ignored by the consumers and if there's a minor change that involves a bit of insight into the matter.

Semantic Versioning Engrosses Semantically Important Changes [4]

As ontology consists of different components- concepts, properties, rules, annotations, axioms, individuals and restrictions, changes made to these components must be analyzed effectively during SemVer in ontology. As ontologies are semantically rich based on these semantics, inferences are deduced .Inferences that are legal in an ontology are important,

both the direct ones in ontology and the indirect ones in some other ontology or system. OWL is monotonic in nature, so deletions are more important than additions. Some changes made are absolutely safe with respect to inference. Changing the asserted (i.e explicit) axioms in ontology are a bit like changing the implementation of some interface rather than its explicit, public, contracted behavior. Here, entire focus is on effectual changes rather than on each change.

Versioning an OWL ontology based on the asserted axioms provides too little flexibility to producers and is tedious task for consumers as well. Therefore, it can be stated: a major change is when there is a change in direct or valid inference (removal), a minor change occurs when indirect inferences are removed, may break SPARQL(Protocol and RDF Query Language) queries, axioms, rules etc. or when direct or indirect inferences are added and a patch change occurs with the change in non-logical parts. Deletions in OWL ontologies should always trigger a major field change in the version identifier as removing inferences are more serious activity than adding them.

Finally, some of the OWL features are non logical which means that they are not semantically significant and therefore do not affect inferences in valid OWL system like RDF(Resource Description Framework) comments, OWL axioms, RDFS(Resource Description Framework Schema) labels, axiom annotations etc. These changes would affect patch field of the SemVer scheme. Renaming a class, in case that does not have axioms or any other ontology is not dependent on it, would not change the major field as no inferences are drawn from it. Trivial changes like addition of the concept, removal etc. would always increment the major field but that scenario is not liked by the programmers as each trivial change would increase the major field each time. There's a restriction that major field is increased only when direct inferences are affected and indirect inferences like importing a class of another ontology would change the minor field.

Cooking ontology in fig 2 is created in protégé (http://protege.stanford.edu/) to show SemVer in OWL ontologies. The graph shown in fig 2 built in OntoGraph tab of protégé exemplifies the hierarchical relationships in cooking domain between different classes. Ontologies are based on the concept of multiple inheritances. In the constructed cooking ontology, there is a total of 34 subclass axioms count, two classes-food and recipes are created. A total of 36 classes are created including, food and recipe. The rest 34 classes are the subclasses of these two classes.

Axioms, data properties, object properties, individuals are created to create semantically rich ontologies. Inferences are deduced and reasoner assists in creating a consistent ontology in protégé.

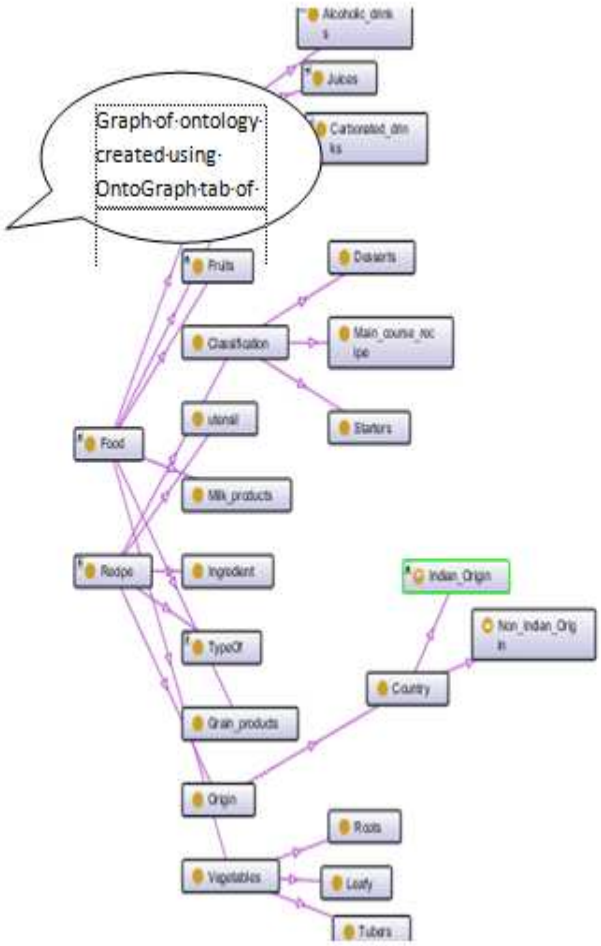

Figure 2: Shows the Hierarchical Relationships between Concepts in Cooking Domain

The induced changes in the ontology affect the different parts of the SemVer scheme. The patch number changes due to change in non-logical parts of OWL ontologies, any change in annotations, comments, labels would affect the patch number. In the fig 1, a part of cooking ontology is shown having beverages as the subclass of food class and a comment "Beverages are the liquid foods" is used to describe what beverages are. In fig 2, this comment is changed to "Beverages are the foods we drink", this change in the comment is a non-logical change and semantically insignificant leading to the change in the patch number. This change never breaks the dependencies/compatibility. If cooking ontology is currently at version 6.2.5, this change would lea to the bump in the patch field 6.2.6. Likewise, any change in labels, annotations, OWL axioms etc., all non logical parts affect the patch number.

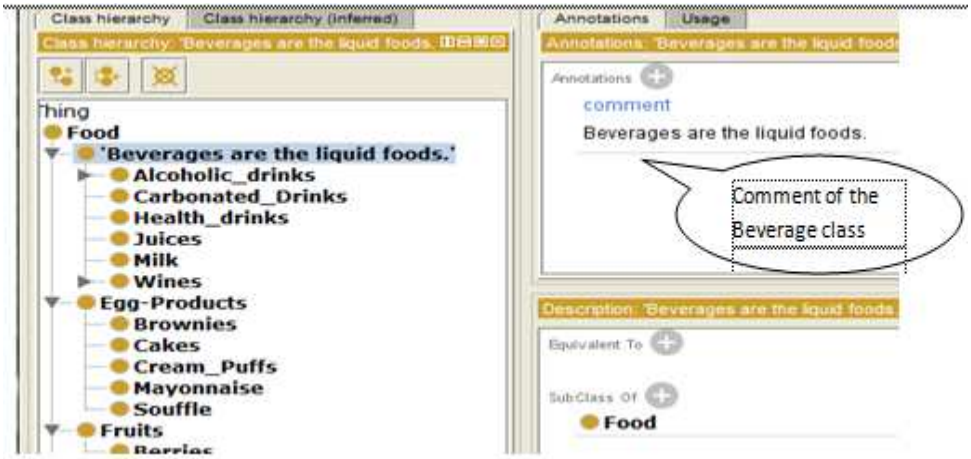

Figure 3: Shows the Part of the Food Ontology with Beverages Class and the Comment

Figure 4: Shows the Change in the Beverages Comment Leading to the Patch Number Change

The figure 3 and 4 shows minor version change in cooking ontology, a new subclass of class fruits, melon, is created. Addition of a new concept is backward compatible with the consumers of the OWL ontology but providers probably been broken. The addition of new class results in the formation of 6.3.0. If an ontology imports another ontology, in the fig 5, cooking ontology imports another ontology food.owl ()leading to creation of all new ontology with addition of new inferences. This is also a minor version change as users does not have to necessarily use the new inferences, they are still compatible with the old version of ontology,as it supports their queries as well. This addition probably breaks the providers of OWL ontology because they need to implement this new change. Fig 5 shows the new cooking ontology shaped by importing another ontology food.owl in cooking.owl.

Figure 5: Show the Class Fruits with Its Subclasses Berries, Citric Fruits and Tropical Fruits

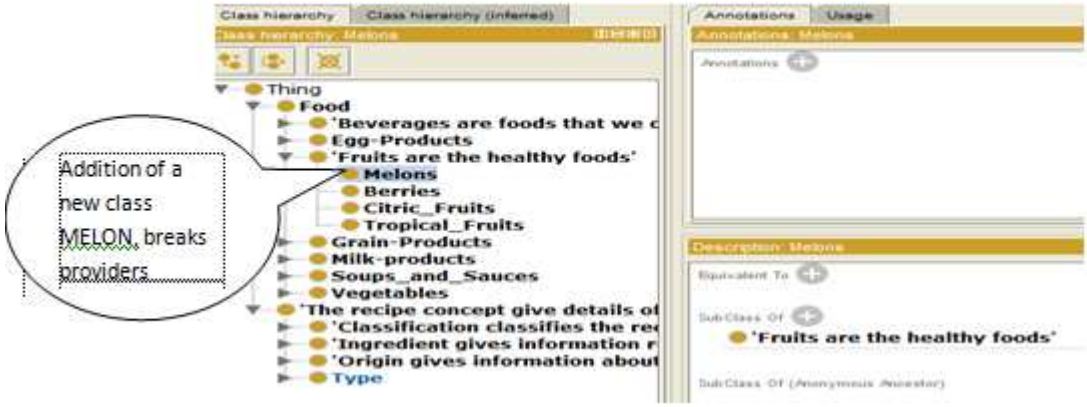

Figure 6: Show the Addition of the New Subclass MELONS in the Fruit Hierarchy (Minor Version Change)

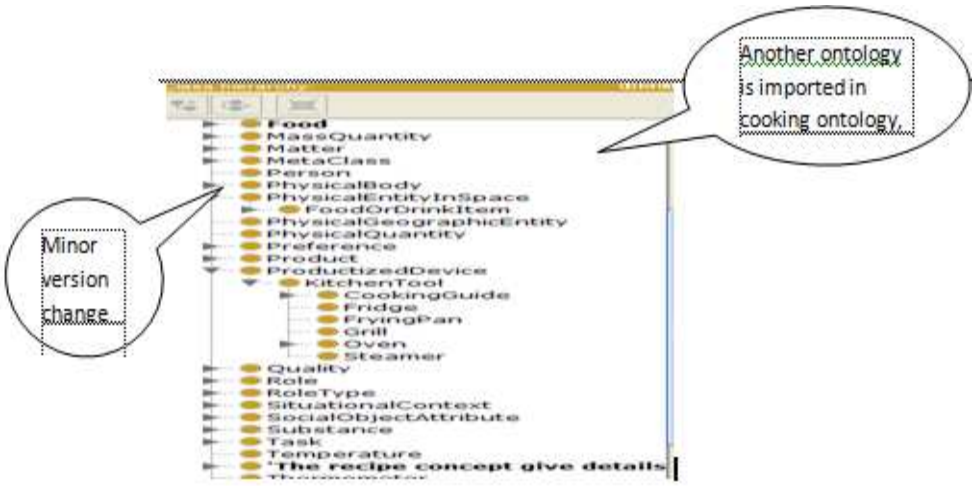

Figure 7: Show the New Cooking Ontology in Which another Ontology Has Been Imported

Major version change breaks both consumers and providers of owl ontology. Deletions play a pivotal role in

bumping the major version number. Major version number bumps when direct inferences are removed. A supposition is made here. Suppose cooking ontology is at version number 6.4.0 after importing the food ontology into the cooking ontology. Users of the new cooking ontology draw inferences from both cooking as well as food ontology. At this point, a change is induced in the new cooking ontology, removing the imported food ontology and deleting its inferences.This change would lead to the breaking of the owl ontology and will lead to the creation of new version 7.0.0. The users of the new cooking ontology will not be compatible with this change as it will break owl ontology users dependent on the inferences drawn from both the ontology. Only the users which were dependent on cooking ontology solely would not break. Those consumers will remain backward compatible, fig 7 shows the cooking ontology when food ontology concepts are removed.

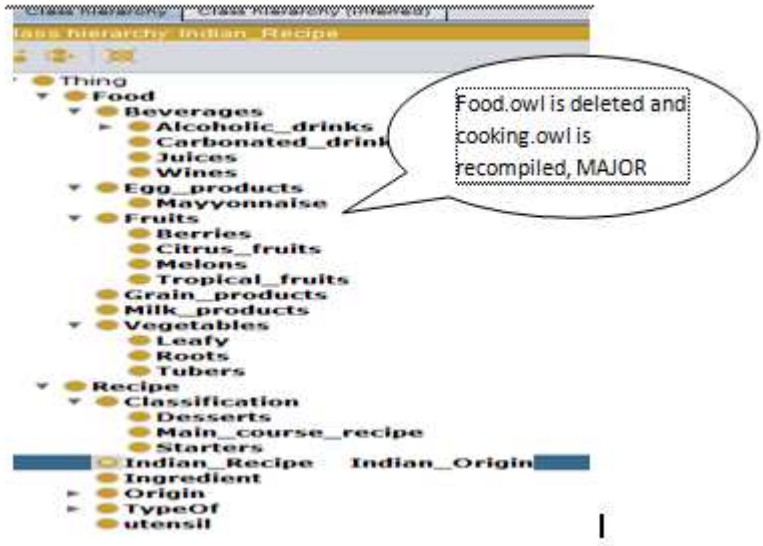

Figure 8: Shows Cooking Ontology after Removal of Food Ontology Concepts

CONCLUSIONS

To allow applications to grow even larger, it is paramount that the versions have semantics and are therefore predictive [3].Semantic versioning has gained importance recently and lend a hand in overcoming the harms of version lock and version promiscuity [6]. It's a way of communicating compatibility issues to the users of the software- providers and consumers. Producers of the software can easily communicate to consumers the kind of changes induced in the software and Consumers can easily identify their compatibility status as soon as a new version is brought into the market. Associating meaning to the versioning scheme avoids the dependency hell nightmare. Semantic versioning scheme is applied to semantically rich cooking ontology created in protégé and a kind of change induced can be judged through the segment affected. The implication here is that ontologies are not similar to public APIs and declaring the software as a public API is the foremost necessity for using SemVer. Ontologies are treated as public APIs and are semantically versioned just as the APIs. In this paper, an attempt has been made to apply the essence of semantic versioning to semantically rich ontology. It requires a further insight in future to make SemVer approach more pertinent to semantically rich ontologies.

REFERENCES

1. Samreen, Shabana, J. S. Mirza, and Anila Rasheed. "RDF and OWL Ontology Building of Web Applications." Research Journal of Information Technology 5.4 (2013): 109-117.
2. Warner T. “Semantic Versioning-2.0.0” Available at :http://www.princeton.edu/semantic_versioning-2.0.0
3. OSGi Alliance, May 6, 2010 “ Semantic Versioning – Technical whitepaper ” Revision 1.0
4. Clark and Parsia “Semantic Versioning”, 2011, Available at: http://weblog.clarkparsia.com/2011/09/semantic_versioning
5. Kempka M.,(2013,June),EclipseSource“Semantic Versioning for Eclipse Developers” Available at: http://www.eclipsesource.com/blogs/2013.
6. Ginnivan, J.,Latest revision(2014, july)“Introduction to semantic versioning”Available at: http://github,com/ParticularLabs/GitVersion
7. Eclipse version numbering, revised in 2009, Available at: http://wiki.eclipse.org/Version_Numbering.
8. Mahajan A, Kaur P “A review on ontology evolution and versioning”,IOSR JCE, Volume 17, Issue 2, Ver. III (Mar – Apr. 2015)PP 35-43
9. Spinellis, D. (2005). Version control systems. Software, IEEE, 22(5), 108-109.
10. Jaziri, W. (2009, October). A methodology for ontology evolution and versioning. In Advances in Semantic Processing, 2009., SEMAPRO'09. Third International Conference on (pp. 15-21). IEEE.
11. Noy, N. F., & Klein, M. (2004). Ontology evolution: Not the same as schema evolution. Knowledge and information systems, 6(4), 428-440
12. Khattak, A. M., Batool, R., Pervez, Z., Khan, A. M., & Lee, S. Y. (2013). Ontology evolution and challenges. J. Inf. Sci. Eng, 29, 851-871.14
13. Stojanovic, L. (2004). Methods and tools for ontology evolution.
14. Chandrasekaran, B., Josephson, J. R., &Benjamins, V. R. (1999). What are ontologies, and why do we need them?. IEEE Intelligent systems, 14(1), 20-26.
15. Hull, D., & Drummond, N. (2005). A Practical Introduction to Ontologies & OWL. Tutorial at the 6th International XML Summer School
16. Noy, N. F., &McGuinness, D. L. (2001). Ontology development 101: A guide to creating your first ontology.
17. Crash course on Protégé ,Petrkremen
18. Sertkaya, B. (2009). Ontocomp: A protege plugin for completing owl ontologies. In The Semantic Web: Research and Applications (pp. 898-902). Springer Berlin Heidelberg.
19. Fuchs, Christian, et al. "Cooking cake." ICCBR 2009 Workshop Proceedings. Vol. 9. 2009.
20. Nanba, Hidetsugu, et al. "Construction of a cooking ontology from cooking recipes and patents." Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication. ACM, 2014.

21. Mota, Sergio Gutiérrez, and Belen Diaz Agudo. "ACook: Recipe adaptation using ontologies, case-based reasoning systems and knowledge discovery."Proceedings of the Cooking With Computers workshop. 2012.

22. Dufour-Lussier, Valmi, et al. "Improving case retrieval by enrichment of the domain ontology." Case-Based Reasoning Research and Development. Springer Berlin Heidelberg, 2011. 62-76.