***Domain-Specific Language for Software Version Numbering***

Sergii Shmarkatiuk, Danny Dig

Department of Electrical Engineering and Computer Science, Oregon State University

{shmarkas,digd}@eecs.oregonstate.edu

*Abstract*—TODO

Keywords—Software Configuration Management; Version Control; Software Version Numbering, Domain-Specific Language, Release Management, DevOps

# Introduction

Discipline of Software Configuration Management studies changes in large and complex systems with the goal to prevent the chaos caused by the numerous corrections, extensions, and adaptations. In order to maintain scalability and continuous delivery of software applications software companies invest in infrastructure and DevOps departments more that ever before. Even though there is a lot of research has been done in the area of Software Configuration Management and number of DevOps practitioners increases, such important problem as unambiguous software versions numbering remains unresolved. Although software developers use software version numbers as a common way to represent corrections, extensions, and adaptations, it turns out that there is no consistent approach of assigning version numbers to new software builds and releases. As a result, people involved in Software Development (Dev) and Release Management (Ops) tend to misunderstand each other as long as there is no way of consistent and non-ambiguous assignment of unique identifiers to new software builds and releases. For example, Release Management team (Ops) often faces situation when Software Development team (Dev) hands over two different software artifacts have the same version (e.g. 1.0.2) even though artifacts have different feature sets. As a result, it becomes difficult to automate release activities without additional information or additional set of conventions.

**P1: Trees instead of graphs (version numbering anthrophobia).** Example: book chapters. Does this really happen? Books do not need to migrate things, while bugfixes should be merged. Tree is a wrong metaphor. Use example from OSS project and DSL presentation slides. Analyze Top 10, look for ‘x’-s. Look for conversion from book chapters to versions with ‘x’s. How far do they go with resolving version numbering inconsistencies? Add pictures and examples.

**P2: Different artifacts with the same name (ambiguity a.k.a identity crisis, version numbering schizophrenia)**. For example, github repository on date of Jan 12th labeled their version 1.1.1, then 2000 LOC came in, but version is still the same 1.1.1. OSS project example. Discussion in the forum about version confusion. Is it resolved? Are these bugs that cannot be replicated? We found similar cases in top 10 most popular repositories throughout their history.

**P3: inconsistency (version number megalomania)**. Firefox and other OSS projects (Eclipse) as an example + 1 or 2 proprietary. Make sure there is full picture and there are no packages with intermediate versions. Painful to introduce automation, introduction of reference/dependency bugs.

Now we present our solution. Does it disambiguate, does it resolve inconsistency? Is it scalable, is it expressive (can it do everything that people did before) and complete? To prove take existing version number history and map it to my DSL version tree. Take at least 3 projects.

We contacted N projects to introduce this version numbering. They confirmed problems with version numbering and our version numbering elegantly solves those problems.

Create survey.

Our study shows that common version numbering practices treat software histories as trees instead of graphs. As a result, software version numbers turn out to be inconsistent, ambiguous, and unreliable and cause confusion among members of software development teams. In order to address the problem of consistent non-ambiguous software version numbering we formalized concept of Software Version in the form of Domain-Specific Language to represent software histories as graphs instead of trees. Our study makes following contributions:

### Demonstrates limitations of existing version numbering approaches.

### Describes novel Domain-Specific Language (DSL) for Software Version Numbering (VN)

### Describes how proposed approach addresses limitations of existing version numbering approaches.

### Shows that proposed approach has potential to become golden standard in order to reconcile software histories that use conflicting version numbering schemes.

# Ease of Use

TODO

# Proposed solution

## Basic definitions

* *Artifact* – programmable entity (examples: class, software component, software project, etc)
* *Number* – non-negative integer that is used to refer to sequential changes of programmable entity (*artifact*) in the scope of the same *dimension*. Represents non-editable state of the *artifact*.

Examples: **0**, **1**, **2**, …

Set: N+ (natural numbers with zero)

* *NumberPlaceholder* – entity that represents editable state of the *artifact*. We use special character ‘**x**’ to represent concept of *NumberPlaceholder* if it is usedas a part of *VersionNumber*.
* Set:  (universal set)
* *VersionCompound* – entity that represents both editable and non-editable states of the *artifact.*

Examples: **x**, **0**, **1**, **2**, …

Set: VC =  cup \!\, N+

* *Dimension* – level of changes granularity. Uses elements of *VersionCompound* set to address different levels of granularity. We use special character ‘**.**’ to represent concept of *Dimension* if it is usedas a part of *VersionNumber*.

Set: D

* *VersionNumber* – composition of *VersionCompounds* to represent unique numeric identifier of programmable entity (*artifact*) using concept of *Dimensions*.

Set: VN = D × VC

Examples: **x**, **0**, **2**, **x.1**, **3.x**, **5.x.x**, **0.x.3**, **x.1.x.23**, …

* *MaturityLevel* – label representing reliability of programmable entity (*artifact*) in terms of estimated expectancy/threshold of errors generated by programmable entity (*artifact*)

Set: ML = {1 => Branch, 2 => Dev, 3 => Test, 4 => User, 5 => ReleaseCandidate, 6 => Production, 7 => Stable}

* *Version* – unique identifier of programmable entity (*artifact*) with regard to possible maturity levels.

Set: V = VN × ML

Examples: **Branch/1.x.x**, **Branch/x**, **Branch/3.0.x** **Dev/0.x.1**, **Dev/5.0.x.2**, **Test/0.x.1**, **User/1.x.0**, **Test/2.4.3**, **User/2.4.3**, **ReleaseCandidate/3.0.6**, **Production/4.1.34**, **Stable/25.3.89.23**

# Graphical representation of version trees

<http://bit.ly/1AxFcCg>

# Toolset

TODO

# Evaluation

TODO

# Conclusion

TODO

##### Acknowledgment

TODO

##### References

1. Phillips, Sillito, and Walker, “Branching and Merging.”
2. Brindescu et al., “How Do Centralized and Distributed Version Control Systems Impact Software Changes?”.
3. “The Commit Size Distribution of Open Source Software.”
4. Alali, Kagdi, and Maletic, “What’s a Typical Commit?”.